THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Selection of Components in Compositional Verification of Safety Properties

Jinjian Shi

This thesis is submitted in partial fulfillment of the requirements for the Degree of Master of Science at the University of Waikato.

July 2009

Abstract

This report presents a two-step components selection method to compose the components for compositional verification. This method employs different methods for the selection of components to be composed during compositional verification. Also, this report presents some automata transformation strategies to improve the efficiency of composing and projection. This enhanced compositional verification method is applied to a set of large and complex realistic industrial examples to evaluate and compare the performance of different methods for components selection. The example *profisafe_i6* [MM03] [MM02] [PN02], which was never verified for the language inclusion check before, is verified first by this enhanced compositional verification method.

Contents

Chapter 1

Introduction

Safety is always the first issue which should be considered when designing a system. In a software system, the safety and reliability of software are the most important quality characteristics [MJD99]. Especially in industry, a precise safety-concerned design may save a lot of time, money, and even life. In order to design the system right and make the system reliable and safe to be used, there is a need to verify the model of the system. Verification is a measure to check the safety of the model and ensure the model is built correctly. According to Balci, "Verification deals with building the model right" [OB97], this is the objective of the term "verification". Therefore, verification of safety properties is crucial and necessary for system design in order to build a right system. However, with increasing requirements and competition, systems become more powerful, more functional, and more effective. Meanwhile, with the development and upgrade of systems, their numbers of components are increasing and their complexities are mounting up, and the trend will continue in the future. This makes verification become a challenge because of the larger and more complex systems.

Usually, a model checker for verification constructs a synchronous product of the model of a system in order to verify it. But it may encounter a problem of state-space explosion for models with a large number of components when constructing a synchronous product. This may cause the model checker to run out of time or out of memory.

The Native model checker [BBFLPP98], [JS06] and BDD model checker [BBFLPP98], [REB86] are two model checkers which construct the synchronous product by composing all components of the model. Both of them encounter the problem of state-space explosion easily, even though the BDD model checker represents the state space symbolically in order to save space.

The Modular model checker [SW07], [BMM04] cuts down the state space consuming by analyzing the synchronous product of subsets of the components of the model instead of the synchronous product of the entire model. So are compositional verification and incremental verification [BMM04], [AFF02]. These three methods obtain improvements for large verification, but the problem of state-space explosion problem still exists.

The Projecting model checker [SW07], [WM08] simplifies the size of the model by projecting some events out and then reducing the state space of synchronous product. This method enhances extremely the performance of verification for large models, but it still has some potential to be improved.

This report presents an enhanced compositional verification method for language projection to reduce the state-space explosion problem. It uses a two-step method to select a set of automata, which is called a candidate, of the system to be composed. The first step is constructing a series of candidates based on the method mustL [FM06]. The second step is selecting the best candidate from the series of candidates according to a certain heuristic which could be maxL, minS, minT [FM06], or minCut. MinCut is first introduced in this report. After the candidate is chosen, projection is activated to compose all of the automata in the candidate into one automaton and project some events out to reduce the size and

complexity of this new automaton.

In order to improve the compositional verification of safety properties, and project more events, a strategy named Automata Conversion [SW07], [WM08] is applied. As for the controllability check, this strategy converts the controllability problem into the language inclusion problem. Instead of checking whether uncontrollable events are enabled at a bad time, it only checks if a certain event can ever happen. This strategy converts all specification automata into plant automata by adding some selfloops and creates a new singleton-state specification automaton which declares what uncontrollable events cannot happen. Ideally, after composing and projection, the final model will have one state if the system is not controllable or zero states if the system is controllable. As for the language inclusion check, this strategy is still working by converting the property automaton into plant and creating a new singleton-state property automaton which tells what bad events can never happen, too.

Furthermore, more improvements are made for enhancing the compositional method. In this project, subsumption test is adopted to remove some unnecessary candidates from the candidates list and reduce the size of it; Selfloop check, Same Transition check, and Almost Same Transition Check are also introduced in order to minimize the size of automata further.

Since projection is introduced in this project, the counterexample is not original any more. Some of the events are projected. Therefore, a backward solution [SW07] is taken to extend the counterexample trace. Also, Almost Same Transition Check could change the trace, but will not change the length of the trace. Hence, the counterexample after Almost Same Transition Check needs to be corrected, too.

This enhanced compositional verification method is applied to a set of large and complex realistic industrial examples to evaluate and compare the performance of four different methods for components selection, and implemented as a toolkit of WATERS [WS]. The example *profisafe_i6* [MM03] [MM02] [PN02], which was never verified for the language inclusion check before, is verified first by this enhanced compositional verification measure.

Chapter 2 presents some of the required knowledge for understanding of this thesis. Chapter 3 describes compositional verification and its algorithm. Chapter 4 presents the strategies of Automata Conversion and Automata Simplification. Chapter 5 discusses the method to construct the candidate list, and the heuristics employed for candidate selection in compositional verification and compares the performance of them according to the experimental data. Chapter 6 presents the conclusion of this project.

Chapter 2

Preliminaries

2.1 Automata Theory

Automata theory is widely used in computer science for analyzing and verifying the finite state model of a system. It is the study of a theoretical model of software systems or computer hardware [MS05].

2.1.1 Automaton

An automaton is a mathematical model for a finite-state machine. Usually, the aim of it is to study the capabilities and limitations of computing processes. A finite-state automaton (finite-state machine) is a machine that, given an input of symbols, jumps from one state to another according to the action of transitions. Formally, a finite-state automaton is described as a tuple [BBFLPP98]

$$A = (Q, \Sigma, T, q_0, Q_m)$$

with

finite set of states	$Q = \{q_1, q_2, q_3, \ldots\}$
finite set of events	$\Sigma = \{a, b, c, \ldots\}$
transition relation	$T \subseteq Q \times E \times Q$

initial state	$q_0 \in Q$
set of marked states	$Q_m \subseteq Q$

For example: the following is an example automaton.

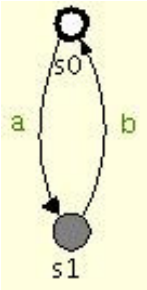


Figure 2.1: An example automaton

According to Figure 2.1 and the definition of automaton, it displays that the finite state set Q is $\{s_0, s_1\}$; the finite event set Σ is $\{a, b\}$; there are two transitions which are $s_0 \rightarrow s_1$ labeled with event a and $s_1 \rightarrow s_0$ labeled with event b; the initial state is s_0 which is an element of Q; and the marked state set is $\{s_1\}$ which is a subset of Q. Putting these information together, it makes up the complete automaton.

2.1.2 Event

In an automaton, events are used to connect states. An event may be moving up or down of an elevator, arrival or leaving of a train, termination of a computer program, start or finish of a task in a manufacturing factory, and so on.

An event is an occurrence or an action affecting the state change of a finite-state

automaton [KG95]. For instance, in Figure 2.1, both "*a*" and "*b*" are events. State s_0 can go to state s_1 when event *a* happens in state s_0. And state s_1 can go back to state s_0 when event *b* happens in state s_1. There is one important thing for event is hidden selfloop. Events which are not described in an automaton can happen at any time and anywhere, but it will not affect the change of state. For example, event *c* is not mentioned in Figure 2.1 above,

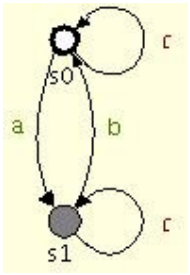


Figure 2.2: An example of Hidden Selfloop

So *c* is not in set *E* but it can happen at any states anytime (both at state s_0 and s_1) as showed in Figure 2.2. Nevertheless, the happening of event *c* does not change the state. Then *c* is a hidden selfloop event and the transition labeled with *c* is a hidden selfloop transition. These selfloops are implicitly added to all states.

2.1.3 Transition

A transition is a passage from one state to another in an automaton [BBFLPP98]. It contains a source state, an event and a target state. For example, $s_0 \rightarrow s_1$ labeled with event *a* and $s_1 \rightarrow s_0$ labeled with event *b* are two transitions.

2.1.4 Automata Type

In this project, there are three types of automata which are plant automaton, specification automaton, and property automaton.

A plant automaton is an automaton representing a model of the system to be controlled. It describes all of the possible behaviors of the model [BBFLPP98].

A specification automaton is an automaton designed to control the plant. It restricts the behavior of the plant by dynamically disabling some of the controllable events to avoid some undesired uncontrollable events happen [BBFLPP98]. So, it describes the desired behaviors.

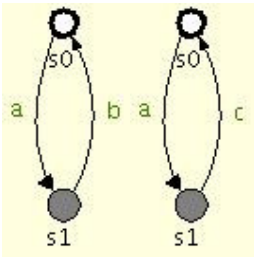
A property automaton is an automaton which states that something bad should never happen [BBFLPP98]. For example, an elevator will never stop at the middle of two floors.

2.2 Synchronous Product

Systems are modeled using more than one automaton in real life, and they interact with each other when they run in parallel. When automata run in parallel, they all run at the same time. Therefore, the events can happen only if they can be accepted by all automata. In order to verify a system, it is needed to find a way to represent the entire system running in parallel. The synchronous product [BBFLPP98] is such a way to represent the system and the interaction of its components.

The synchronous product describes the current state of the whole system with a state tuple $q = (q_1, q_2, \ldots q_n)$ [BBFLPP98].

For example, there are two automata:



A1 A2

Figure 2.3: Two example automata

At first, both of them are at initial state, so the system composed by these two automata can be represented as (s_0, s_0). In Figure 2.3, there are three available events which are event *a*, *b*, and *c*. As for automaton A_1, according to the definitions of selfloop, event *a* and *c* are enabled in the initial state while event *b* is not enabled, because event *b* is mentioned in this automaton but only enabled in state s_1. And for automaton A_2, event *a* and *b* are enabled while event *c* is not enabled for the similar reason with event *b* in A_1. Hence, as for the entire system, only event *a* is enabled in the initial state by all automata. When event *a* happens, the system state goes to (s_1, s_1). In this state, both event *b* and *c* are enabled by the system, but not event *a* because it is only enabled by all automata in the initial state. Therefore, when event *b* happens, the current system state becomes (s_0, s_1) from (s_1, s_1); and when event *c* happens, the current system state becomes (s_1, s_0) from (s_1, s_1). Likewise the system can return to (s_0, s_0) from (s_0, s_1) when event *c*

happens and return to (s_0, s_0) from (s_1, s_0) when event b happens. In the example above, the system representation (s_0, s_0), (s_1, s_1), (s_0, s_1), (s_1, s_0) are the synchronous product states. The complete synchronous product is displayed in Figure 2.4:

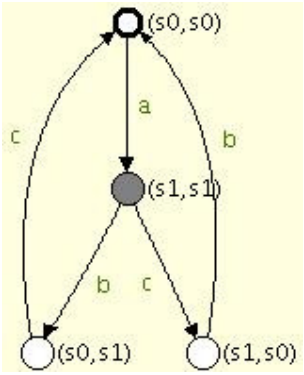


Figure 2.4: An example Synchronous Product

2.3 Safety Property

A safety property is a property indicating that something bad can never happen [BBFLPP98]. It aims to keep the system safe. In this project, there are two safety properties, which are controllability and language inclusion.

2.3.1 Controllability

In general, there are two types of event which are controllable events and uncontrollable

events. A controllable event is an event which can be disabled and enabled. An uncontrollable event is an event which cannot be prevented from happening.

Let P (Plant) and S (Specification) be two automata. S is called controllable with respect to P if, for every state (q_p, q_s) reachable in the synchronous product of P and S, every uncontrollable event e which is enabled in q_p also is enabled in q_s [RW89].

As for controllability, plant automata and specification automata are considered only. The specifications send controllable events to the plants and tell the plants what to do, while the plants send back uncontrollable events to the specifications and tell the specifications what has happened to them. The idea of designing the specification is to limit the behaviors of the plant in order to make the plant behave as desired. However, is it possible that the specification automata can limit the behaviors of the plant automata? In order to know the possibility, it is needed to check the controllability of the system which includes both the plants and the specifications. If this system is controllable, then the specifications can limit the behaviors of the plants; that is to say that for any reachable states of the plants, if an uncontrollable event can happen, then the specifications should also allow this uncontrollable event to happen. Otherwise, if this system is not controllable, that is to say that an uncontrollable event is enabled in the plants but it is not enabled in the specifications, and then the specifications cannot limit the behaviors of the plants because an uncontrollable event cannot be prevented from happening.

For example, as for the system in Figure 2.3, assume that A_1 is a plant automaton and A_2 is a specification automaton. Then in state (s_0, s_1), A_1 allows event a and event c to happen while A_2 only allows one event out of these two events to happen which is event c. Therefore, if event a is an uncontrollable event, then this

system is not controllable because specification A_2 disables an uncontrollable event a while it cannot be disabled.

2.3.2 Language Inclusion

This project only considers deterministic discrete event systems (DES). The behavior of a deterministic DES can be described by the set of sequences of events [RW89]:

$$\sigma_0 \sigma_1 \ldots \sigma_n$$

and the initial state s_0. One of these kinds of sequences of events is called a trace or string of the system, and a collection of traces is called a language.

Language inclusion is used to check if the language of a model is included by the language of a given automaton [BMM04]. This given automaton is a property automaton. For example, if and only if every trace in the language of a set of automata A is also in the language of property automaton B, then A satisfies B.

As for language inclusion, plant automata and specification automata are treated as the same automata which are non-property automata. The algorithm to check if the non-property automata satisfy the property automaton is to construct the synchronous product of both of the non-property automata and the property automaton, and check that if whenever all non-property automata enable an event, the property automaton will enable this event too. If this event is always enabled, then this property automaton is satisfied.

2.3.3 Counterexample

A counterexample is a sequence of events which a specific instance of the falsity of a model. The counterexample trace represents that a bad event can happen eventually. If a model is failed to pass the verification of safety properties, then the model checker will automatically compute such a counterexample. [BBFLPP98]

2.4 Projection

There exist some events that are used exclusively by only one automaton or a set of automata, or can be removed away for other reasons in some large systems composed of several automata. As a result, the alphabet Σ is partitioned into the set Y of events to be removed away and the set Ω of events to be retained. Typically, Y consists of the events used exclusively by the automata considered.

Natural projection

$$P_{\Sigma \rightarrow \Omega}: \Sigma^* \rightarrow \Omega^*$$

is the operation that removes all events not in Ω from a string [RW89]. This operation can be extended to operate on languages as well.

Inverse projection

$$P^{-1}{}_{\Sigma \leftarrow \Omega}: 2^{\Omega^*} \rightarrow 2^{\Sigma^*}$$

inserts events into all strings at all possible positions [RW89].

According to Simon Ware's report, projection and inverse projection can also be

applied to automata. Projection is implemented by replacing all occurrences of the events to be hidden (the events in γ) by the *silent* event τ first, and then using a determinisation algorithm [HMU01] to make the resultant nondeterministic automaton deterministic. Inverse projection is achieved by adding selfloops with the hidden events to all the states of the automaton. [WM08]

2.5 Model Checker

There are three different kinds of model checkers used in this project to check controllability and language inclusion of the models. These three model checkers are Native model checker [BBFLPP98], [JS06], BDD model checker [BBFLPP98], [REB86] and modular model checker [SW07], [BMM04]. These three model checkers are available in WATERS [WS].

2.5.1 Native Model Checker

The Native model checker is one of the monolithic model checkers and is implemented in Java and refined in C++. It constructs the synchronous product of the model to be checked and explores all of the states of the synchronous product to verify the model. This method describes the state space of the synchronous product of the model explicitly and the memory used to represent the synchronous product directly depends on number of states.

2.5.2 BDD Model Checker

The BDD model checker is a BDD-based model checker and is implemented in Java. BDD means binary decision diagram. A BDD is an ordered graph representation of a Boolean function. The BDD model checker is also one of the monolithic model checkers which constructs the synchronous product of the model to be checked and explores all of the states of the synchronous product. The difference is that the BDD model checker represents the state space symbolically with BDD nodes instead of listing the state space explicitly. The number of BDD nodes is usually smaller than the number of states.

2.5.3 Modular Model Checker

The modular model checker analyses the subsets of the automata of a model instead of checking the entire set of automata of the model at the same time. Since the synchronous product of a model is the intersection of all languages of the language of individual component of the model, if a subset of the specifications of a model is checked to be controllable with respect to a subset of plants of the model, then the entire system cannot contain a counterexample for the subset of specifications and plants. Therefore, it is possible to verify that a specification is controllable or a property is satisfied without composing the entire set of automata of the model.

2.6 Examples for Evaluation

A computer with an AMD3200+ (2GHz) CPU and a 2GB RAM is used for testing of this project. The following are the examples used for evaluation. The models of

these examples are available in WATERS [WS].

- big_bmw is the model of the BMW E65 CAS window lift controller [PD00] [PM03]. bmw_fh is a small version of it.
- ftechnik, fzelle are about case studies of two different production cells [LL95] [LM96].
- rhone_alps, and rhone_tough describe an AIP automated manufacturing system [BC94] [RJL96].
- tbed_ctct, tbed_nocoll, tbed_noderail, tbed_uncont represent a train testbed [RJL96].
- verriegel4, ftuer, and koordwsp represent a central locking system.
- small_factory_2 comes from the classical "small factory" example from [RW89]. bfactory is an upgrade version of small_factory_2.
- tictactoe represents a model of a tic tac toe game.
- mx27 represents a case of a maze game.

The above examples are used for controllability check. small_factory_2, bfactory, bmw_fh, ftuer, koordwsp, tictactoe, and mx27 are small examples and the rest of them are large examples with more automata.

- models beginning with profisafe represent the PROFIsafe field bus protocol [MM03], [MM02], [PN02].

These models are used for language inclusion check. They are extremely large examples. The verification of example *profisafe_i6*, which was never solved before, is solved first in this report.

2.7 Implementation

This enhanced compositional verification method is implemented in Java and available in WATERS [WS]. The information of classes and data structures is available in [WS]. In order to switch different components selection methods efficiently, this implementation allows a methods selection option from the command line. This implementation also allows options for projection limit, state limit, model checker, and safety properties.

Chapter 3

Enhanced Compositional Verification

3.1 Introduction

The verification of safety properties is an essential part for ensuring that the model is developed correctly and the model does not contain errors or bugs [CMM05]. Commonly, a model checker composes the entire set of automata of a model to obtain the synchronous product. This can easily cause the state-space explosion problem. Therefore, if a model checker can compose some automata first instead of the entire set of automata of the model, and then simplify the composition, the whole model could become smaller and the state-space explosion problem can be reduced. This strategy is called compositional verification. It composes a subset of the automata of the model first and simplifies the composition with projection [WM08].

One question in compositional verification is how the automata are composed and their events are projected. Different selection strategies and orders can lead to different performance. Hence, the automata selection is a substantial part of compositional verification. This report presents an enhanced compositional verification measure to solve the automata selection problem. The enhanced compositional verification uses a two-step method to select the automata to be composed out of the entire set of automata of the system. One set of selected automata is called a candidate. The candidates are groups of automata sharing the

events that can be removed because they are not used anywhere else but in this group of automata only. The first step is constructing a series of candidates according to a method called mustL [FM06]. The second step is selecting the most proper candidate from the series of candidates according to certain heuristics such as maxL, minS, minT [FM06], and minCut. The candidates construction method and candidates selection methods are discussed in detail in chapter 5. After the candidate is chosen, projection is employed to compose all of the automata in the candidate into one new automaton and project some events out to reduce the size and complexity of this new automaton. This new automaton is added in the model and the set of automata which are composed are removed from the model. Then the model is changed and has a new set of automata. This new set of automata can be composed and projected again with the two-step candidate selection method. In this way, the compositional model verifier can iteratively compose automata of the model and project some events out. Finally, the modified model is passed to a non-projection safety model checker for safety verification.

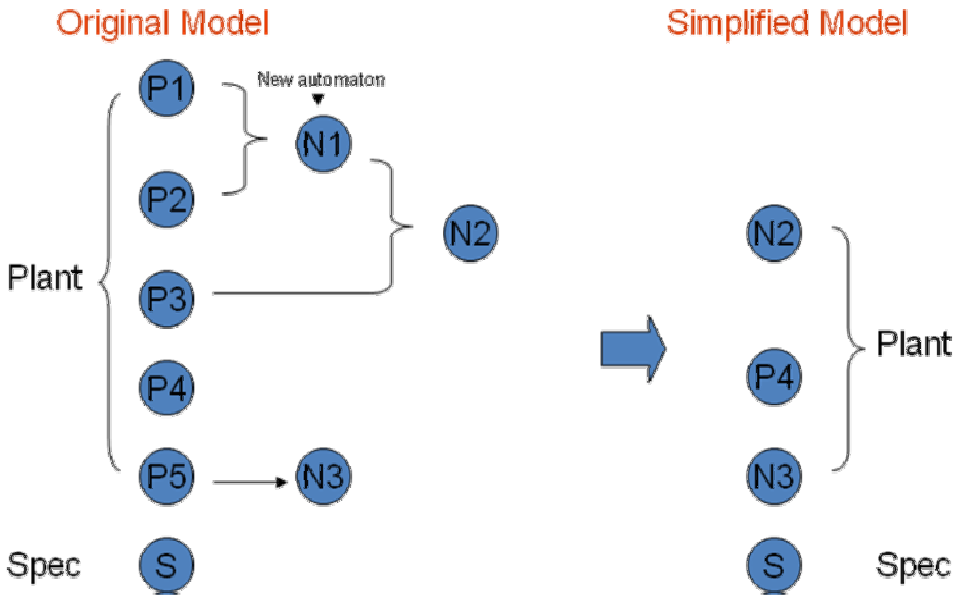


Figure 3.1 Simple process of Enhanced Compositional Verification

Figure 3.1 displays a simple example process of enhanced compositional verification. There is a model with six automata which are plant *P1*, *P2*, *P3*, *P4*, *P5* and specification *S*. With enhanced compositional verification, *P1* and *P2* are chosen at the first round, because they share some events which are not used anywhere else. These events, called local events [FM06], can be projected out. After composing *P1* and *P2* and projecting the local events, a new automaton *N1* is created. Then *P1* and *P2* are replaced by *N1* and the model has five automata which are *N1*, *P3*, *P4*, *P5* and *S*. At the second round, *N1* and *P3* are selected and they have some local events, too. After composing *N1* and *P3* and projecting out the local events, *N2* is created. Then the model has four automata which are *N2*, *P4*, *P5* and *S*. At the last round, there is only one candidate which is *P5*. *P5* has one or more local events which can be projected. So *P5* can be simplified just by

projecting the local events out and becomes to a new automaton *N3*. Then the original model becomes the simplified model which has four automata which are *N2*, *P4*, *N3* and *S*. Then this simplified model can be verified by a non-projection safety model checker.

3.2 Algorithm

The following is the enhanced compositional verification algorithm based on Simon Ware's projection algorithm [WM08]. In this algorithm, the hidden events set is the set of all events mentioned by plant automata only because compositional verification composes plant automata only. The reason is that wrong results may be produced if certain events from the property or specification automata are projected out during compositional verification. A candidate is a set of automata from which a set of events *E* can be projected because all events in *E* are local and not used anywhere else.

Let *Plant* be the plant set of the model. Let *Events* be the event set of the model. Let *Hidden* be the set of event that can be projected in the model.

1. With *Plant* and *Hidden* as input, find every candidate according to the candidates construction method. Add these candidates into a list of candidates *C*.
2. If *C* is empty, return.
3. Select a proper candidate *can* with a set of local events *E* from *C* according to a given heuristic.
4. Compose and project *can*. If the maximum state limit for the projection is not exceeded, then a new automaton *aut* is obtained. Otherwise remove *can* from *C* and go to 2.
5. Replace the automata of *can* by *aut* in *Plant*. Remove *E* from *Hidden* and *Events*.

Figure 3.2 Candidate Selection and Projection Algorithm

Figure 3.2 presents an algorithm of candidate selection and projection. The events set *E* is a nonempty set of events and a subset of *Hidden*. According to the algorithm in Figure 3.2, the events in *Hidden* are not necessarily removed completely. That is because there is a state limit for the projection. If the number of states of the new composed automaton exceeds the limit, this candidate is not a proper candidate for reducing the size of the model. Therefore, this candidate is dropped and the events supposed to be hidden by this candidate are not hidden. The given heuristics mentioned at step 3 in Figure 3.2 are heuristic maxL, minS,

minT, and minCut.

As for the whole model, the compositional way can be used iteratively in order to make the model as small as possible. The following is the iterative compositional verification algorithm. This algorithm assumes that all events, which are not mentioned by specification automata for controllability check or not mentioned by property automata for language inclusion check, can be projected.

Let *Plant* be the plant set and *Spec* be the specification set of the model. Let *Events* be the event set of the model.

1. Let *Hidden* be the set of events used in *Plant* only.
2. With *Plant* and *Hidden* as input, do the composing and projection using the algorithm in Figure 3.2 and get the new *Plant*, new *Events* and new *Hidden*.
3. If *Plant* and *Hidden* are not changed, go to 4. Else go to 1.
4. Create a new model *M* with automata set *Spec* and *Plant*, and events set *Events*.
5. Send *M* to a non-projection safety model checker for analyzing.

Figure 3.3 Iterative Compositional Verification Algorithm

Figure 3.3 presents an iterative algorithm of compositional verification. The idea of this iterative algorithm is continually composing a subset of the automata of the model first and projecting the local events of it to obtain a new automaton, and then replacing this set of automata with the new automaton. Also, for every time

of iteration, the set of automata of the model is different. Hence, the candidate list needs to be reconstructed for every time of iteration.

3.3 Counterexample Extension for Projection

The enhanced compositional verification in this report applies projection to improve the performance of verification. Projection simplifies automata by removing some events. However, this method has its drawback. With projection, some events which should in the counterexample trace may be projected, and the trace may not be complete any longer.

If a model has a counterexample trace which contains an event σ before projection, the model must also have the counterexample trace which contains the event σ after projection. However, this event σ could be removed from the model by projection, and the counterexample trace obtained from the model after projection may not a proper trace for the original model. Therefore, there is a need to extend the trace by inserting the σ event back. Fortunately, according to Simon Ware's study, the short counterexample trace can be extended to a complete trace which is a proper trace for the original model by inserting some of the removed events. The following is Simon's algorithm [SW07] [WM08] to find such a trace.

Let t be a counterexample found in the model made of the events σ_0, σ_1 ... σ_{n-1} when n is the length of the trace t, also let A be the original automaton before it is projected. Let Q be a queue of tuples of states in A, length through t, and built-up trace, and let S be a set of pairs of state and length through t, and Σ' be the set of kept events.

1. Add the tuple (*initialstate*(A),0,[]) to Q and (*initialstate*(A),0) to S.
2. Remove the first tuple (*state*,i,s) in Q.
3. If $i = n$ return s.
4. If there is an outgoing transition from *state* labeled with σi to *next* add (*next*,i+1,sσi) to Q and (*next*,i+1) to S unless (*next*,i+1) $\in S$.
5. For every event α not in Σ', if there is an outgoing transition from *state* labeled with α to *next*, then add (*next*,i,$s\alpha$) to Q and (*next*,i) to S unless (*next*,i) $\in S$.
6. Go to 2.

Figure 3.4: Find Trace Algorithm

This algorithm takes a breadth first search through the original automaton and tries to insert some events which are removed by projection into the projected counterexample trace in order to make this counterexample trace acceptable by the original automaton. In other words, this algorithm runs the incomplete trace in the original automaton and inserts the proper missing events which are from the removed events into the trace. If the current event of the trace is enabled in the current state of the automaton, it remains the current event of the trace. Otherwise, if all events are tried in the current state, try the events which are hidden. Find all hidden events which are enabled in the current state and try each of them until

reach an event in the incomplete trace. Those events appear only in this automaton and do not appear in any other automata, so they can be inserted anywhere into the trace and other automata cannot reject the trace.

If the model is iteratively projected, the following algorithm is adopted to find a proper counterexample trace.

Let A_1, …, A_n be the automata simplified by iterative projection, and let Σ'_i be the set of events A_i is projected with respect to, and let t be the projected trace.

1. Set i to equal n.
2. If i equals 0 return t.
3. Find the trace s using the algorithm in Figure 3.4 with A_i and Σ'_i as input.
4. Set the new value of t to s, and decrement i, then go to 2.

Figure 3.5: Iterative Find Trace Algorithm

This algorithm is based on the algorithm in Figure 3.4. It uses a backward strategy to extend the trace. It starts from the last automaton, which is projected to the first automaton. This algorithm needs to know which set of events are projected from which set of automata.

Chapter 4

Automata Transformation

Compositional verification with projection can effectively reduce the state-space explosion problem [RW89], but there are still some models which are too large to be verified. In order to reduce the state-space explosion problem further and take more advantage of compositional verification, it is necessary to employ more strategies to cut down the size of models.

This chapter presents two types of strategies to improve the performance of projection and reduce the size of automata. These two types of strategies are Automata Conversion [SW07], [WM08] and Automata Simplification. Automata Conversion converts specifications into plants for controllability check, and properties into non-properties for language inclusion check to remove some events, which are used in specifications or properties. Automata Simplification tries to simplify the individual automaton by removing or replacing some unnecessary events.

Also, the counterexample trace is changed and not accepted by the original model anymore after Automata Conversion and Almost Same Transition Check in Automata Simplification. This chapter describes some methods to correct the trace.

4.1 Automata Conversion

4.1.1 Introduction

According to Simon Ware's paper, Automata Conversion converts the model into a new model which has more events that can be hidden and reduce the state-space explosion problem further by adding some selfloop events. Furthermore, after Automata Conversion, the model is easier for safety verification.

Simon Ware [SW07], [WM08] presents a strategy of translating the controllability problem into a language inclusion problem for the standard controllability check. As for controllability check, Automata Conversion converts the standard controllability problem into a language inclusion problem. In this way, this strategy not only can project more controllable events which are used in specifications, but also converts the problem of checking if uncontrollable events can happen at an improper time into a much simpler problem of checking if certain bad events can happen [SW07].

Since Automata Conversion works for controllability check, it should also have the probability of working for language inclusion check. Therefore, this project presents an Automata Conversion method for language inclusion check. As for language inclusion check, Automata Conversion converts the language inclusion problem into another language inclusion problem. The difference between these

two problems is that more events can be projected in the model after Automata Conversion.

4.1.2 Automata Conversion for Controllability Check

As for controllability check, Simon Ware's method converts the plant P and the specification S in the original model into a plant P' and a specification S' by adding new γ events into the model such that for all the specifications S_i and uncontrollable events v_j, the event γ_{ij} can happen only in a situation where v_j is allowed by P but not allowed by S_i [SW07], [WM08]. The following describes how to convert the specification automata and plant automata.

4.1.2.1 Specification Automata Conversion

For every uncontrollable event v_j and for all states s in specification S_i, if there is no outgoing transition labeled with v_j in s, add the selfloop transition (s, γ_{ij}, s). In a specification automaton, the γ events that are added have nothing to do with other specification automata.

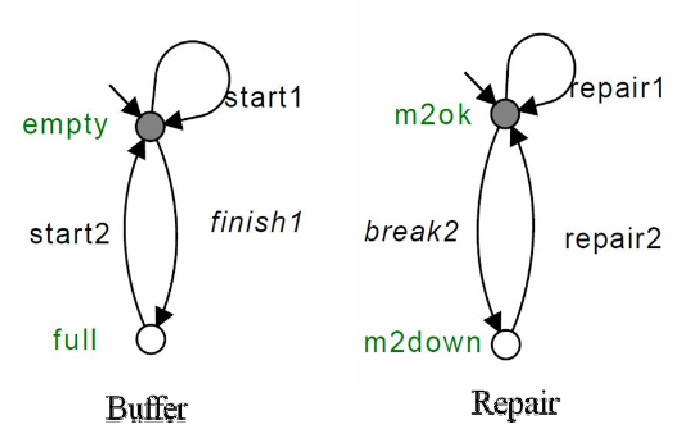


Figure 4.1: Specification Buffer and Repair

Figure 4.1 displays two specifications of the model *small_factory_2* which are *Buffer* and *Repair*. This model comes from the classical "small factory" example from [RW89]. In specification *Buffer*, there is only one uncontrollable event *finish1*. In specification *Repair*, there is only one uncontrollable event *break2*. The rest of the events in these two specification automata are controllable. Then, according to the specification Automata Conversion algorithm above, these two specifications are converted to the modified automata displayed in Figure 4.2.

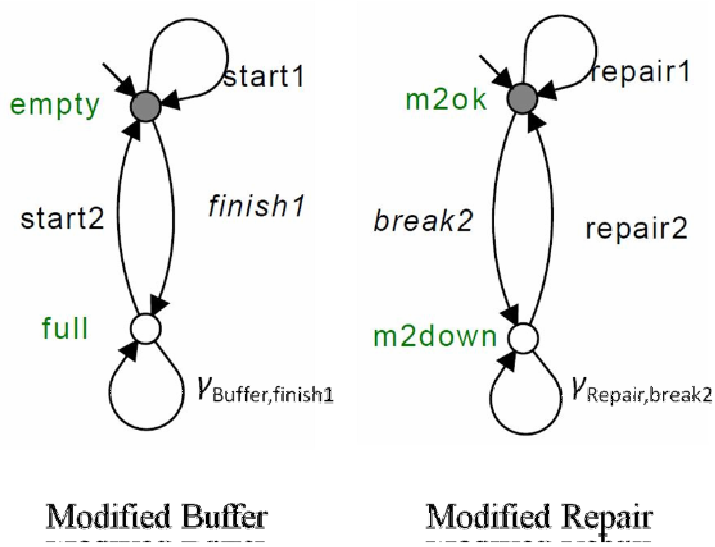


Figure 4.2: Modified Specification Buffer and Repair

In *Buffer*, there are two states which are *empty* and *full*, and one uncontrollable event *finish1*. In state *empty*, there exists an outgoing transition (*empty*, *finish1*, *full*) labeled with *finish1*. So the γ event is not added in this state. But in state *full*, there is no outgoing transitions labeled with uncontrollable event *finish1*. Therefore, a selfloop transition (*full*, $\gamma_{\text{Buffer,finish1}}$, *full*) is added to this state.

In *Repair*, there are two states which are *m2ok* and *m2down*, and one uncontrollable event *break2*. As for the uncontrollable event *break2* in state *m2ok*, there exists an outgoing transition (*m2ok*, *break2*, *m2down*) labeled with *break2*. So the γ event is not added in this state. But in state *m2down*, there is no outgoing transition labeled with uncontrollable event *break2*. Therefore, a selfloop transition (*m2down*, $\gamma_{\text{Repair,break2}}$, *m2down*) is added to this state.

4.1.2.2 Plant Automata Conversion

After the conversion of specification automata, a set of γ events are obtained. The

plant automata also need to be converted by adding some γ events in order to make the model equivalent to the original model. The following is the method of plant Automata Conversion.

For every uncontrollable event v_j, which also is mentioned by specifications, and for all states s in plant P_k, if there is an outgoing transition labeled with v_j in s, then add the selfloop transition (s, γ_{ij}, s) for every possible i. Hence, once the plant P_k allows the event v_j, it will also allow any event γ related to v_j [SW07], [WM08].

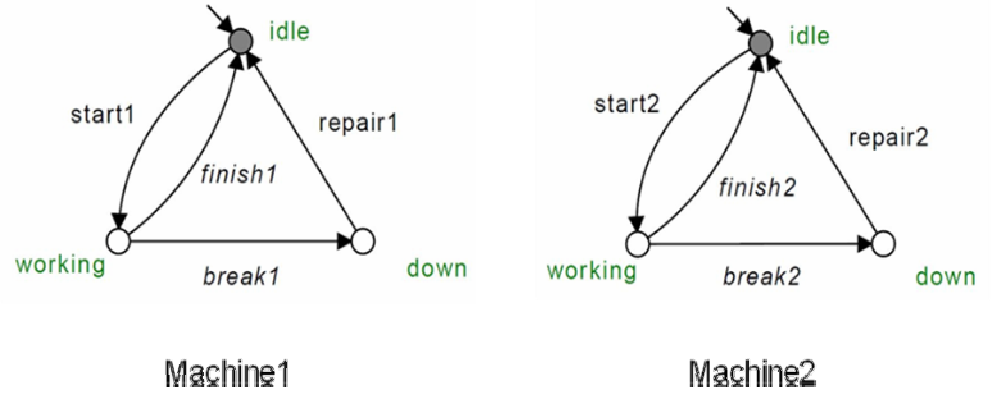


Figure 4.3: Plant Machine1 and Machine2

Figure 4.3 displays two plant automata, which are *Machine1* and *Machine2*, of the model *small_factory_2*. There is one uncontrollable v_j event, which is *finish1*, in plant *Machine1*. Also, in plant *Machine2*, there is one uncontrollable v_j event *break2*. Then, these two plant automata are converted to the following modified automata according to the plant Automata Conversion method.

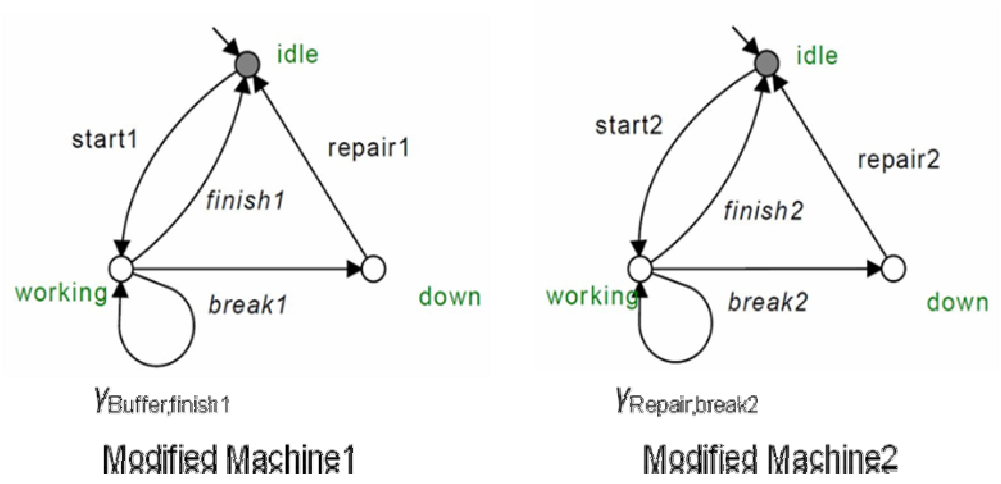


Figure 4.4: Modified Plant Machine1 and Machine2

Figure 4.4 displays the modified plant automata of *Machine1* and *Machine2*. In plant *Machine1*, there are three states, which are *idle*, *working*, and *down*, and one uncontrollable v_j event, which is *finish1*. In state *idle* and *down*, there is no outgoing transition labeled with uncontrollable v_j event *finish1*. Hence, the γ event is not added into these two states. In state *working*, there is an outgoing transition (*working*, *finish1*, *idle*) labeled with uncontrollable v_j event *finish1*. So a selfloop transition (*working*, $\gamma_{Buffer,finish1}$, *working*) is added to this state.

In plant *Machine2*, the situation is similar. In state *idle* and *down*, there is no outgoing transition labeled with uncontrollable v_j event *break2*. Hence, the γ event is not added into these two states. In state *working*, there is an outgoing transition (*working*, *break2*, *down*) labeled with uncontrollable v_j event *break2*. So a selfloop transition (*working*, $\gamma_{Repair,break2}$, *working*) is added to this state.

4.1.2.3 One-state specification creation

After the conversion of specifications and plants, the γ_{ij} events can happen if and

only if the uncontrollable v_j events are enabled in all plants but disabled in specification S_i. Therefore, as for controllability check, the modified specification automata are treated as plant automata and a one-state specification is created for the checking.

The one-state specification automaton contains a single state and a set of blocked events which are the γ events. Figure 4.5 displays such a specification of model *small_factory_2*. The main aim of this one-state specification is to tell the controllability checker that the γ events should not happen. For instance, in Figure 4.5, event $\gamma_{Buffer,finish1}$ and event $\gamma_{Repair,break2}$ are not enabled in any states and can never happen.

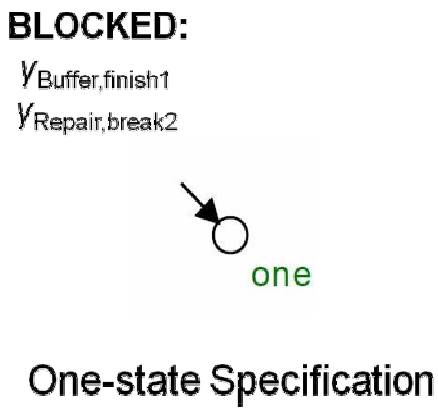


Figure 4.5: An Example One-state Specification

4.1.2.4 Evaluation

This test uses the heuristic minS for the second step of candidate selection and the projection limit is 1000. The non-projection model checker used for the checking

is the Modular Controllability Checker. The examples used in the evaluation are from [PD00], [PM03], [LL95], [LM96], [BC94], [RJL96].

Model	With composing and projection			With model conversion, composing and projection			γ Events	R Events	Original Events
	States	Events	Time(s)	States	Events	Time(s)			
big_bmw	168	66	0.304	6	47	0.683	47	66	66
Ftechnik	374297	117	2.408	24	156	5.479	117	117	117
Fzelle	1448	82	0.670	4	113	1.610	113	82	88
rhone_alps	1540	74	0.416	2	38	0.590	38	74	90
rhone_tough	O	76		366141	74	8.517	40	76	98
tbed_ctct	1057536	88	4.452	O	200		131	88	116
tbed_nocoll	81522	88	1.057	43976	190	15.641	138	88	116
tbed_noderail	1308927	88	5.718	43572	190	14.863	138	88	122
tbed_uncont	470752	88	3.326	8083	190	16.006	138	88	116
small_factory_2	10	6	0.296	2	2	0.264	2	6	8
Bfactory	19	5	0.297	1	2	0.264	2	5	12

Table 4.1: Automata Conversion Results for Controllability Check

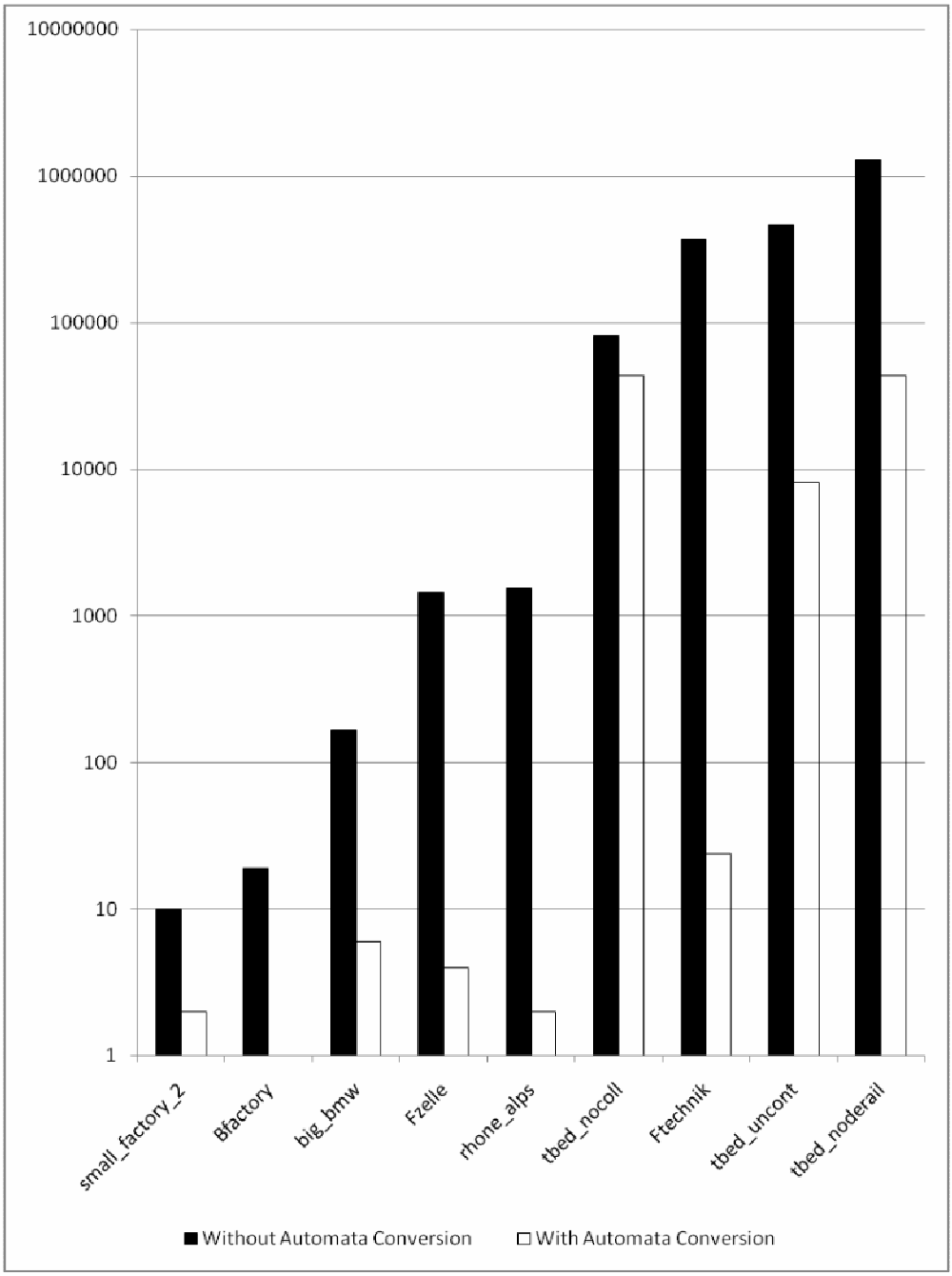


Figure 4.6: Peak number of States after projection

with and without Automata Conversion

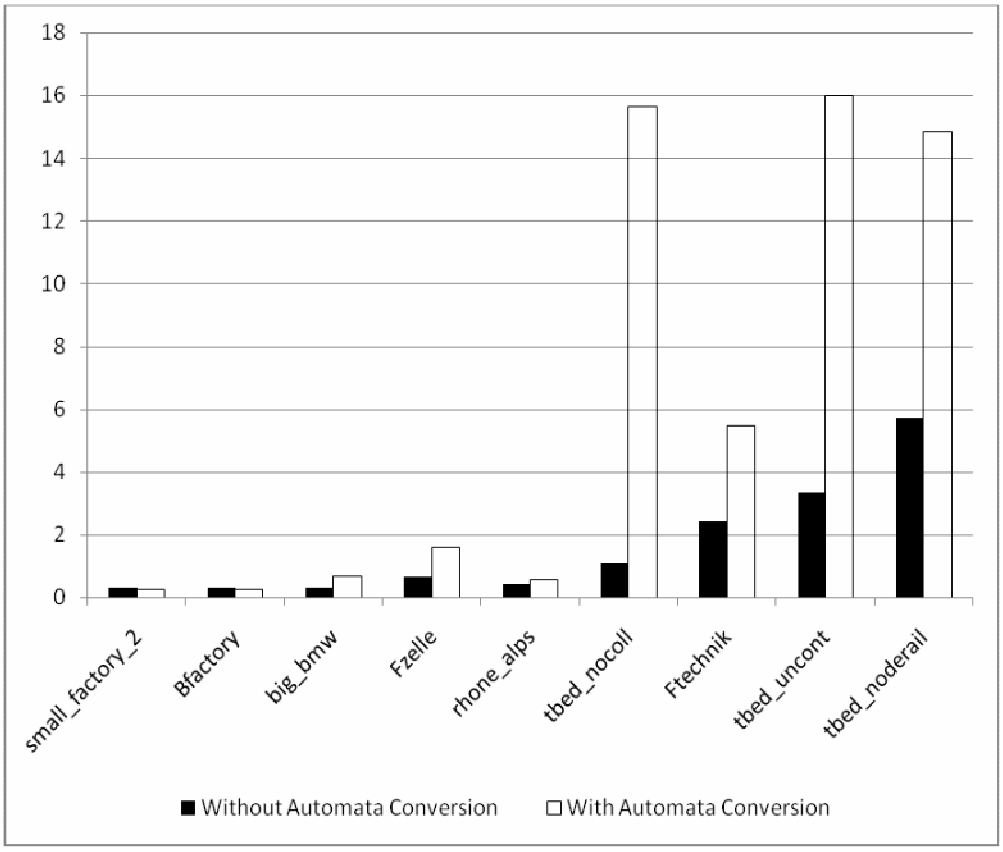


Figure 4.7: Controllablity Check Processing time
with and without Automata Conversion

Table 4.1 displays two test cases. The first case is to check the model with composing and projection. The second case is to check the model with Automata Conversion, composing and projection. The content of the test includes six parts which are States, Events, Time, γ Events, R Events, and Original Events. States is the peak number of the states of the synchronous product of the model after projection that the Modular Controllability Checker explores during the verification. Events is the amount of events of the model after projection. The Time is the processing timeof the model checking. γ Events is the created events during Automata Conversion. R Events is the events that shared by plants and

specifications. Original Events is the events of the model without Automata Conversion.

According to Table 4.1, the synchronous product of the model after projection with Automata Conversion is smaller than the model without Automata Conversion. For example, as for model *ftechnik* [LL95], [LM96], the number of states explored by the model checker without Automata Conversion is 374297, which is fifteen thousand times larger than the number of states explored by the model checker with Automata Conversion which is 24. Furthermore, as for the model *rhone_tough* [BC94], [RJL96], it cannot be solved without Automata Conversion. But it can be solved with Automata Conversion. According to Figure 4.6, with the increase of the peak number of states, the effect of Automata Conversion is much more observable. The peak number of states is much smaller after Automata Conversion. The number of events can also be cut down, if the model is small enough and the local events can be projected mostly or completely. For example, the model *small_factory_2* [RW89] and *bfactory* are quite small and their numbers of events of the model with Automata Conversion, composing and projection are 3 and 2 respectively which are smaller than their numbers of events of the model without Automata Conversion, which are 6 and 5 respectively. In these two models, the numbers of R Events are 6 and 5 respectively, and the numbers of γ Events are 2 and 2 respectively. Therefore, without Automata Conversion, there are at least 6 events remaining for model *small_factory_2* and 5 events remaining for model *bfactory.* With Automata Conversion, there are at least 2 events remaining for mdoel *small_factory_2* and 2 events remaining for model *bfactory.* That is to say, as for model *small_factory_2*, there are four more events which are projected, and as for model *bfactory*, there are three more events which are projected. But often, the number of events increases. That is because the number of events that the Automata Conversion creates is much more than the

number of events that cannot be projected before but can be projected after. For instance, as for model *tbed_nocoll* [RJL96], its γ Events number is 138 while its original events number is 116. That is to say, the number of events created is 138 and the number of events of the model after projection is at least 138, which is larger than the number of events of the original model. Figure 4.7 shows the result of the processing timewith and without Automata Conversion. The model checking processing timeof the model with Automata Conversion is longer than the processing timeof the model without Automata Conversion. The reason of slow checking time is because there are more projection after Automata Conversion and projection is time consuming.

As a result, the Automata Conversion strategy can reduce the state-space problem and project more events for controllability check, but it needs a longer process time. Furthermore, some models can only be solved with Automata Conversion.

4.1.3 Automata Conversion for Language Inclusion Check

Since the controllability problem can be converted into a language inclusion problem and more events can be projected, a language conclusion problem can also has the possibility of being converted into another language inclusion problem in order to remove more events. Instead of removing more controllable events, which are used in specification automata, this strategy removes more events which are enabled in all states of property automata. However, this strategy only suits for the model with a property automaton which has some events that are enabled in all states of the property automaton.

4.1.3.1 Property Automata Conversion

For every event v_j which is not enabled in all states and for all states s in property R_i, if v_j is not enabled in s, then add the selfloop transition (s, γ_{ij}, s) to s.

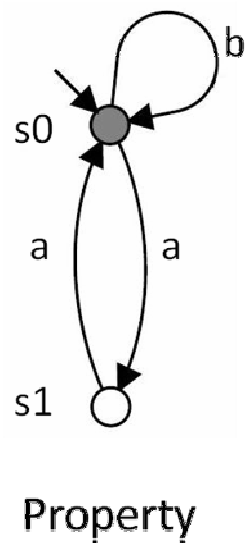


Figure 4.8: An example property

Figure 4.8 displays an example property. This property automaton has two states which are *s0* and *s1*, and two events which are events *a* and *b*. Event *a* is enabled in all states and event *b* is only enabled in state *s0*. According to the property Automata Conversion algorithm, this property is converted into the modified property in Figure 4.9.

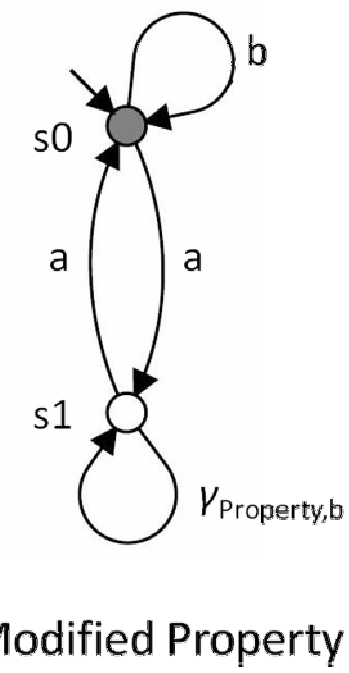


Figure 4.9: Modified Property

In the property automaton *Property*, there is one event *b* which is not enabled in all states. As for event *b*, it is enabled in state *s0*. So the γ event is added in state *s0*. In state *s1*, event *b* is not enabled. So the selfloop transition ($s1$, $\gamma_{\text{Property,b}}$, $s1$) is added to this state.

4.1.3.2 Non-property Automata Conversion

For every event v_j, which is not enabled in all states of the property automaton, and for all states s in non-property automaton NP_i, if event v_j is enabled in s, then add the selfloop transition (s, γ_{ij}, s) for every possible non-property automaton NP_i. Hence, once the non-property automaton NP_i allows the event v_j, it will also allow any event γ related to v_j.

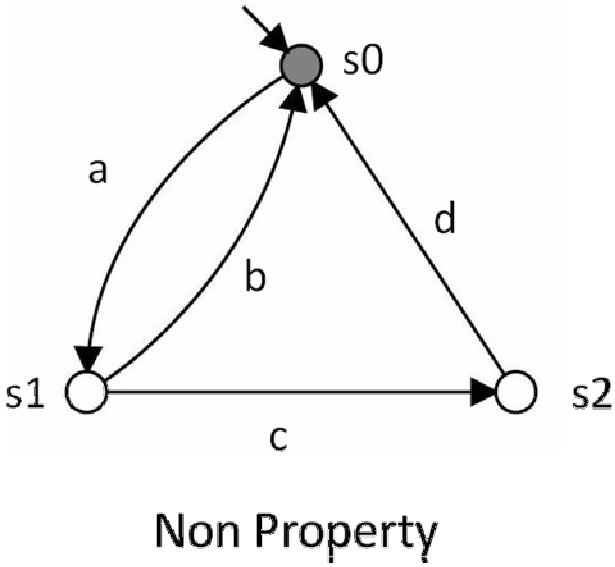


Figure 4.10: An example Non-Property

Figure 4.10 displays an example non-property automaton. This automaton has three states which are *s0*, *s1*, and *s2*, and four events which are event *a*, *b*, *c*, and *d*. Thereinto, event *b* is not enabled in all states of property *Property*. According to the non-property Automata Conversion, the automaton in Figure 4.8 is converted into the modified automaton in Figure 4.11.

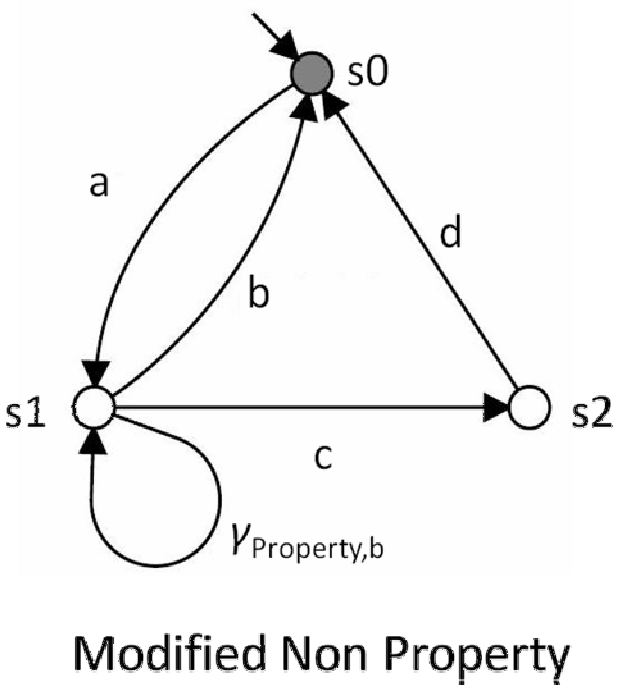


Figure 4.11: Modified Non-Property

In the non-property automaton, there is one event *b* which is not enabled in all states of property *Property*. As for event *b*, it is not enabled in state *s0* and *s2*. So the γ event is not added in state *s0* and *s2*. In state *s1*, event *b* is enabled. So the selfloop transition (*s1*, $\gamma_{Property,b}$, *s1*) is added to this state.

4.1.3.3 One-state property creation

After the conversion of property automata and non-property automata, the modified property automata are treated as non-property automata and a one-state property is created for the checking. The one-state property automaton contains a single state and a set of blocked events which are the γ events. Figure 4.12 displays such a property. In this one-state property, event $\gamma_{Property,b}$ is not enabled in any states and shall never happen.

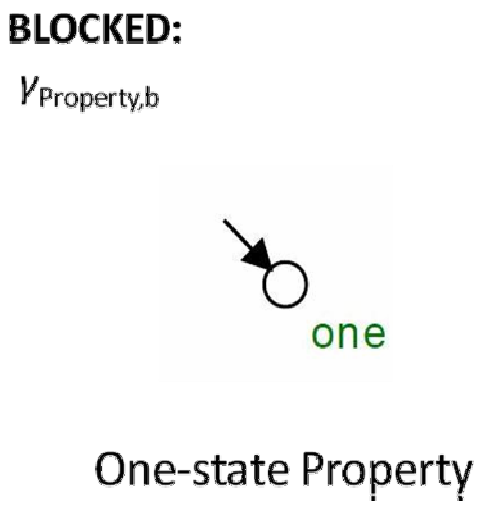


Figure 4.12: An example One-state Property

4.1.3.4 Evaluation

This test takes the heuristic minS for the second step of candidate selection and the projection limit is 1000. The non-projection model checker used for the checking is the BDD Language Inclusion Checker. The examples used in this evaluation are from the PROIsafe field bus protocol [MM03], [MM02], [PN02].

Model	Property	With composing and projection			With model conversion, composing and projection			OE	EE	γE	RE
		Nodes	Events	Time(s)	Nodes	Events	Time(s)				
profisafe_i4_host	HOST__fv_timeout__property	141	4	11.169	1	1	7.192	218	3	1	4
profisafe_i4_slave	SLAVE__fv__property	3642	32	2.825	1	21	1.512	130	11	21	32
profisafe_i4	HOST__fv_crc__property	330116	195	29.543	288624	120	59.734	393	82	1	83
	HOST__fv_crc_noinit__property	1498633	201	90.120	134107	127	198.867	393	83	1	84
	HOST__fv_timeout__property	O	99		29982	32	16.435	378	3	1	4
	SLAVE__fv__property	O	47		32499	37	16.606	378	11	6	17

Table 4.2: Automata Conversion Results for Language Inclusion Check

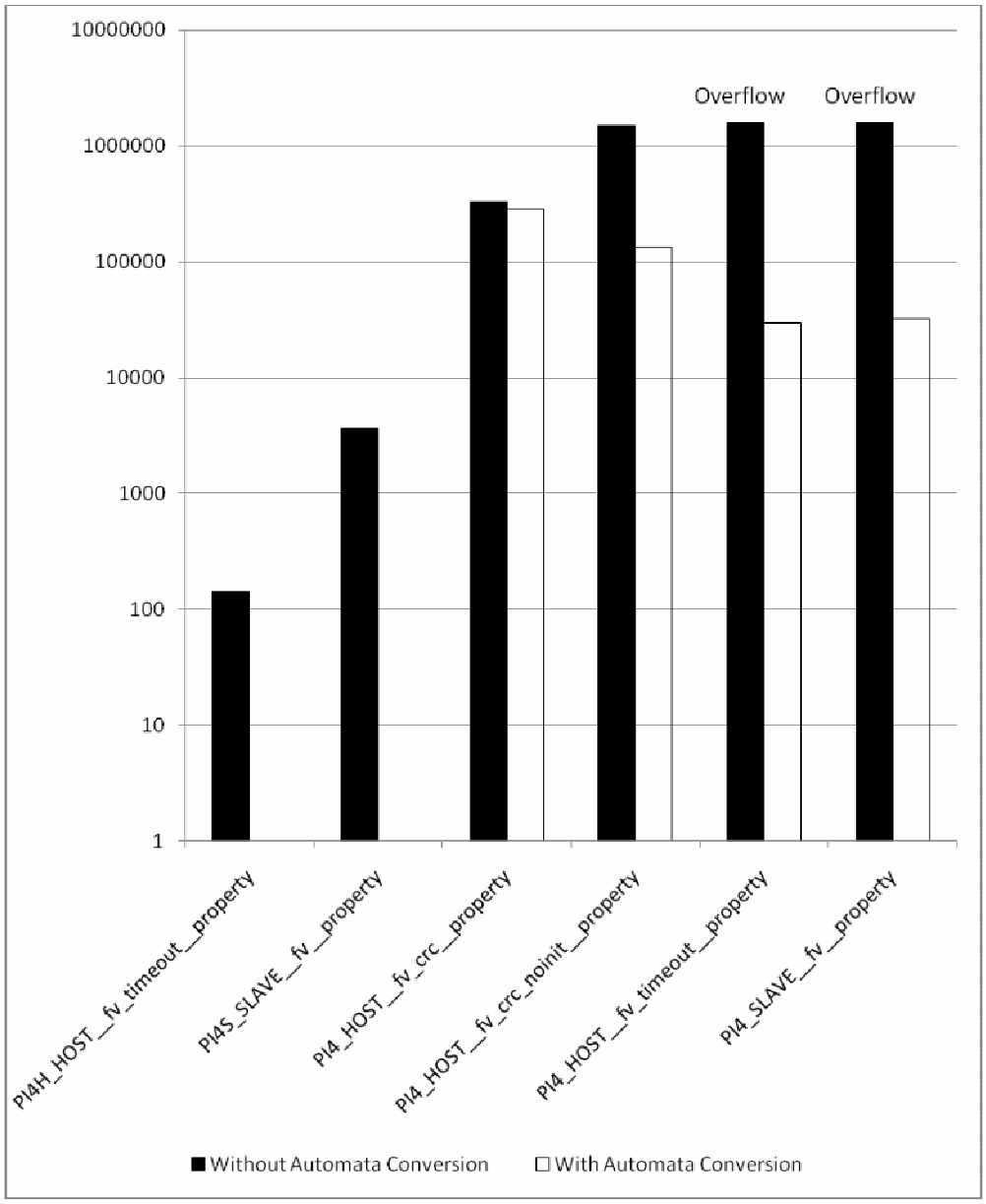


Figure 4.13: Peak number of Nodes after projection

with and without Automata Conversion

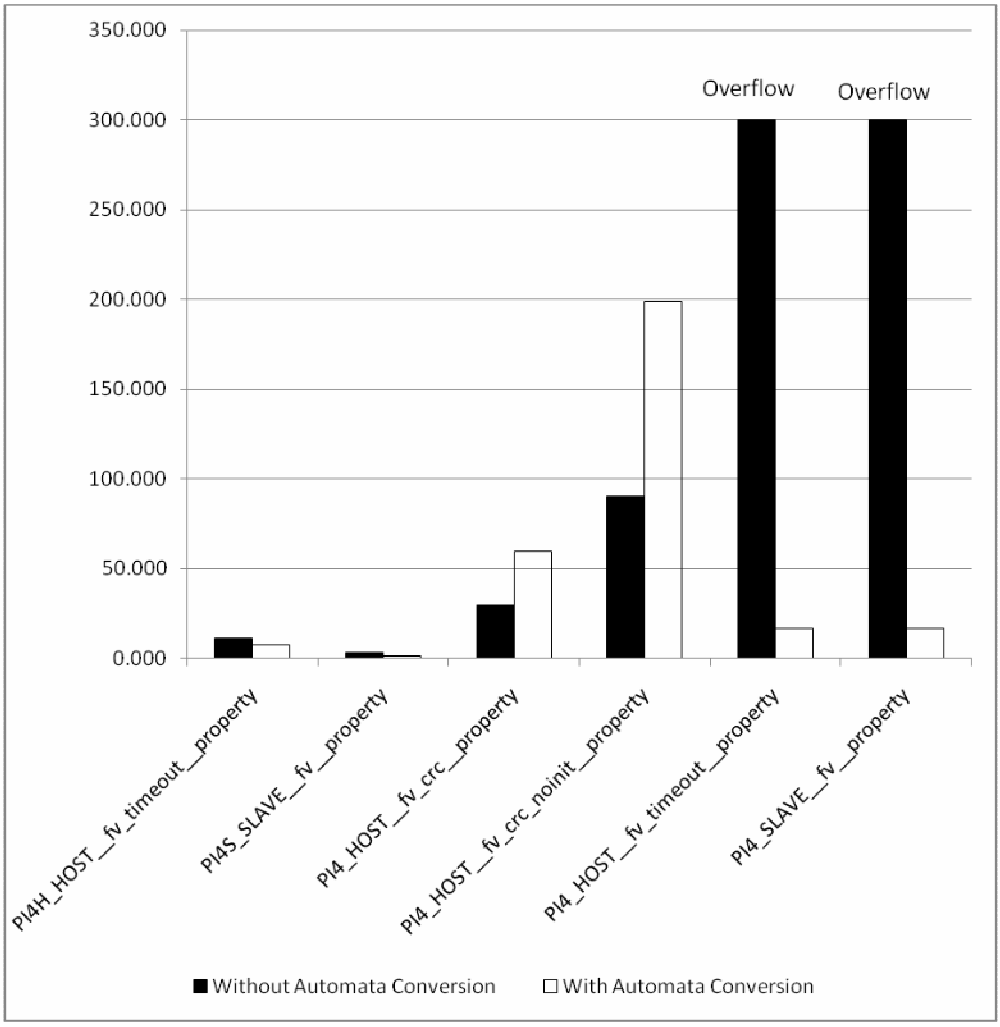


Figure 4.14: Language Inclusion Check Process Time
with and without Automata Conversion

Table 4.2 displays two test cases. The first case is to check the model with composing and projection. The second case is to check the model with Automata Conversion, composing and projection. The content of the test includes three parts which are Nodes, Events, and Time. Nodes is the peak number of the nodes of the synchronous product of the model after projection that the BDD language inclusion checker explores. Events is the number of events of the model after projection. Time is the processing timefor the model checking. Also, Table 4.2

displays other four parts for evaluation which are OE, EE, γE, and RE. OE is the number of original events of the original model. EE is the number of events enabled in all states of the property which can be hidden after the Automata Conversion. γE is the γ events created during Automata Conversion. RE is the related events shared by both property automata and non-property automata. There is only one property in each example.

According to Table 4.2, the peak number of nodes explored by the BDD checker after projection with Automata Conversion is smaller than the peak number of nodes after projection without Automata Conversion. For example, as for model *profisafe_i4_host* with the property *HOST__fv_timeout__property*, the peak nodes number of the synchronous product of the model without Automata Conversion is 139289, which is one hundred and forty thousand times larger than the peak nodes number of the synchronous product of the model with Automata Conversion which is 1. Furthermore, as for the model *profisafe_i4*, it is not solved without Automata Conversion when checking the property *HOST__fv_timeout__property* and the property *SLAVE__fv__property*. But it is solved with Automata Conversion. According to Figure 4.13, Automata Conversion works well with large examples. The events number can be cut down even though there are many γ events that are created. However, the difference of the model checking processing timebetween the check with Automata Conversion and the check without Automata Conversion cannot be defined because of the different situation of the model. If the proportion of time for projection is higher in the entire check process time, then the entire check processing timewithout Automata Conversion is faster than the entire check processing timewith Automata Conversion because the model check with Automata Conversion has more projection.

As a result, the Automata Conversion strategy can reduce the size of the

synchronous product of the model and project more events for language inclusion check. Furthermore, some models can only be solved with Automata Conversion.

4.1.4 Counterexample Correction for Automata Conversion

Automata Conversion can be used to project more events and make the model smaller. But this method has a drawback of modifying the counterexample. The counterexample trace contains one new γ event which cannot be accepted by the original model. Therefore, the counterexample trace needs to be fixed.

The key strategy of automaton conversion is to build a singleton state specification automaton or property automaton with some new uncontrollable γ events which are created according to the original specification automata or property automata. In this singleton state automaton, these new uncontrollable γ events must not happen. In this way, the converted model is failed to pass the safety verification once one of these γ events happens. That is to say, the last event in the counterexample trace must be one of these new uncontrollable γ events. Therefore, the counterexample is changed and there is only one γ event needs to be renovated.

Therefore, the source events, which the creation of new events relies on, need to be remembered when the automaton conversion proceeds. Thus, a map which takes the source events as key and the new events as value is built to track down the original events of the new events. Then, use the original event to replace the new event at the end of the counterexample trace and finish the counterexample correction for automaton transformation.

Let M be the map of storing source event α_i and new event β_i pairs. Let t be the counterexample trace and n be the length of trace t.

1. Find the last event β_i of t which is at position n-1.
2. Find the source event α_i of new event β_i according to M.
3. Replace the new event β_i with the source event α_i in trace t, and return t.

Figure 4.15: Counterexample Correction Algorithm for Automata Conversion

4.2 Automata Simplification

4.2.1 Introduction

All of the experimental data show that, the smaller automata are, the faster composing and projection are executed. Therefore keeping the automata as small as possible is vital not only for composing and projection, but also for verification. Especially for large models, they need composing and projection more than small models because of their huge sizes and complexity. However, many automata are not small and simple enough because of bad design or as a result of projection. For example, after projection, some states are amalgamated into one state. Hence, some transitions which originally have different sources or targets may share the same sources and targets after projection. Then those transitions may become unnecessary and not influence the result of verification. Consequently, they can be removed or replaced in order to keep the automata as small and simple as possible. The following are three aspects to simplify automata.

4.2.2 Selfloop Check

In an automaton, if an event is enabled at all states and used by selfloop only, this event and selfloops can be removed in this automaton.

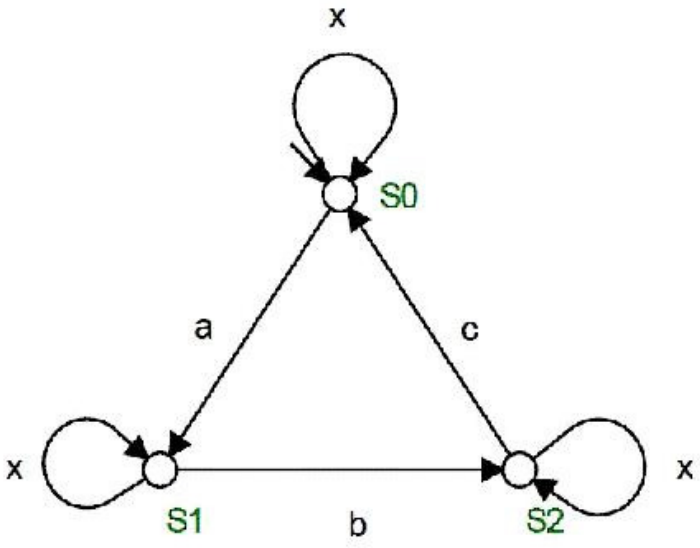


Figure 4.16: An example of Selfloop Check

For the example above, event *x* is enabled in all states and it is removed in this automaton. So are the selfloops labeled with event *x*.

Furthermore, if event *x* only ever is used as selfloop events, and event *x* is controllable or it is used in plants only, then event *x* can be removed, too. The reason is that controllability verification is only interested in uncontrollable events in specifications, and event *x* cannot change the state because it is a selfloop event. Therefore, the result of verification cannot be influenced by removing such an event.

4.2.3 Same Transition Check

If two or more events are only enabled together (they share the source and target) in all automata, they are called same transition. They can be replaced by a single

event.

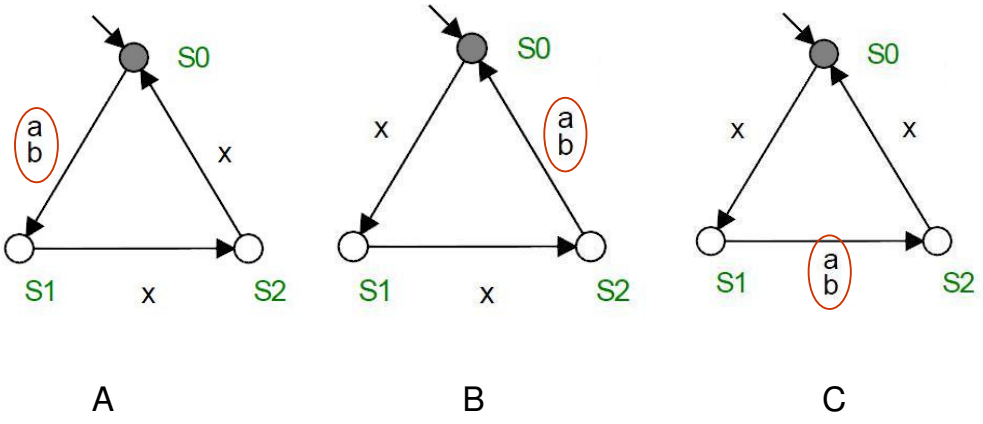


Figure 4.17: An example of Same Transition Check

For the example in Figure 4.17, this model has three automata which are *A*, *B*, and *C*. Event *a* and event *b* are only enabled together in them. Then event *a* and *b* are same transition and they can be replaced by a single event.

4.2.4 Almost Same Transition Check

If two or more events are only enabled together (they share the source and target) in all automata except one automaton, they are called Almost Same Transition. They can be replaced by a single event. The automaton in which the events are not enabled together is called an Almost Same Transition Automaton.

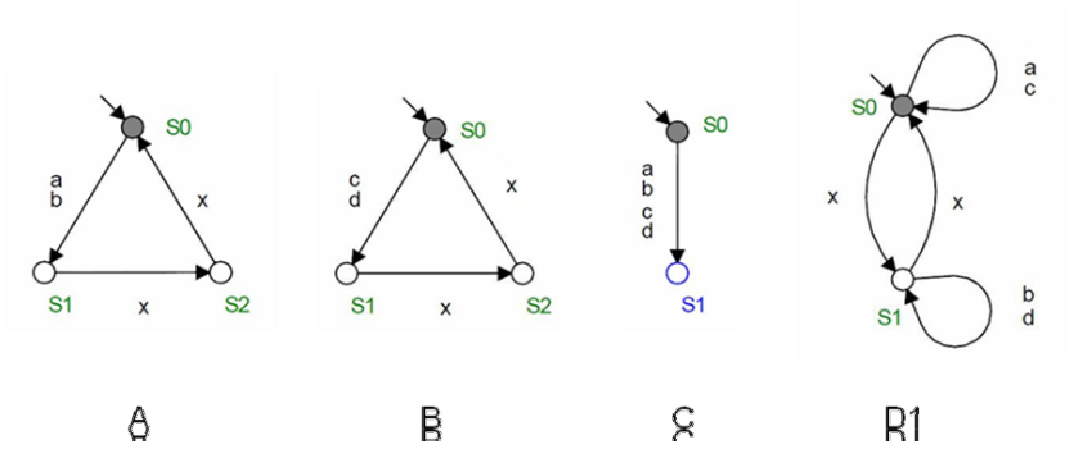


Figure 4.18: An example of Almost Same Transition Check

For the example in Figure 4.18, this model has four automata which are *A*, *B*, *C*, and *D1*. Events *a* and *b* are only enabled together in *A*, *B* (hidden selfloop), and *C* but not *D1*. So they are Almost Same Transition and can be replaced by a single event. So are events *c* and *d*. And *D1* is the Almost Same Transition Automaton for events *a* and *b*. As for events *c* and *d*, *D1* is the Almost Same Transition Automaton, too. As for a real model, the Almost Same Transition Automaton can be different and more than one.

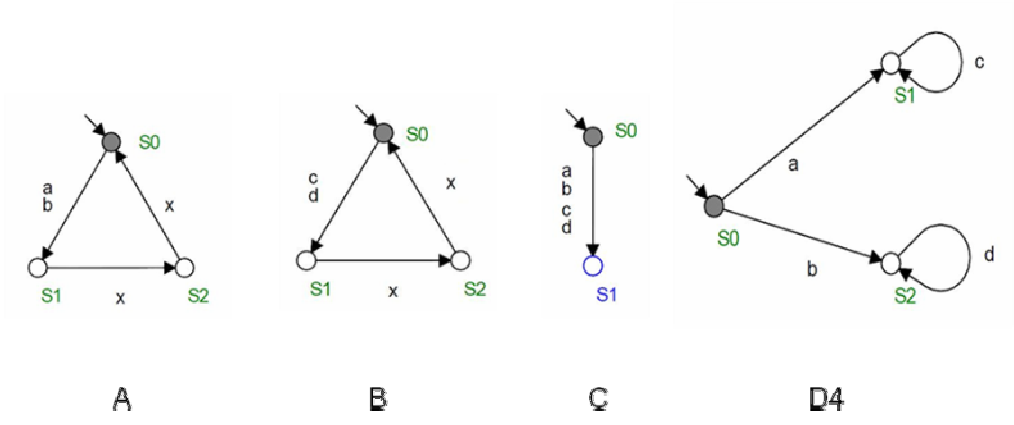


Figure 4.19: Nondeterminism Problem of Almost Same Transition Check

However, if those events share the same source in this particular automaton, they cannot be Almost Same Transition. That is because if they are replaced by a single event, then the automaton will become nondeterministic. For the example in Figure 4.19, events *a* and *b* share the same source state *s0* in automaton *D4*. If they were replaced by a single event *e* for instance, there are two different successor states for state *s0* and event *e*. This is not supported by non-projecting model checkers used in this project. Thus, this case cannot be treated as an instance of Almost Same Transition.

4.2.5 Counterexample Correction for Almost Same Transition Check

Almost Same Transition Check is the only automaton simplification strategy which changes the counterexample trace in the automaton simplification section. The point of Almost Same Transition Check is replacing some events with a new single event in a certain automaton. Hence this strategy does not change the length of the counterexample trace but changes the events of the trace. Also, there may be more than one group of events which can be replaced in one automaton. Therefore, the correction of such a counterexample trace is to find the new events in the trace and replace them with the original events replaced by them.

In order to replace the new events in the counterexample trace with the original events, it is necessary to build a map *M* to store the replacement information. The construction of *M* can be done during the Almost Same Transition Check. For instance, in Figure 4.18, the Almost Same Transition Automaton is *D1* and there are two sets of events replaced in *D1*. If events *a* and *b* are replaced by a single event *e*, and events *c* and *d* are replaced by a single event *f*, the map *M* is

$$M: \quad e \longmapsto \{a, b\}$$
$$f \longmapsto \{c, d\}$$

The following is the algorithm of counterexample correction for events replacement in Almost Same Transition check.

Let t be a counterexample found in the model with Almost Same Transition Check made of the events $\sigma_0, \sigma_1 \ldots \sigma_{n-1}$ when n is the length of the trace t, also let A be the Almost Same Transition Automaton. Let M be the replacement map which stores what new events replace what original events.

1. Set the current state *state* to the initial state of automaton A and i to 0.
2. If σ_i is enabled in *state*, then set current state *state* to the successor of *state* with event σ_i. Go to 4.
3. If σ_i is a new event then look up σ_i in M to find an original event which is enabled in *state*. Set *state* to the successor of *state* with event σ_i.
4. Increase i. If $i = n$ return t.
5. Go to 2.

Figure 4.20: Find Replaced Events Algorithm

This algorithm only works for deterministic models. Also, the Almost Same Transition Check states that the group of events which are replaced cannot occur at the same place and cannot have the same source states, either. Hereby, if the counterexample trace has a new event, this event can be replaced by only one

original event. That is to say, in the current state, if there is an outgoing transition labeled with the new event, there is at least one outgoing transition labeled with the original events which are replaced by the new event. The original events may be more than one, but there is at least one event than can bring the system to the same state as the counterexample.

As for a whole model, there may be more than one Almost Same Transition automaton. Also, the Almost Same Transition Check happens after every projection. The following algorithm is for such a trace correction.

Let A_1, …, A_n be the Almost Same Transition automata, and let M be the replacement map which stores what new events replace what original events, and let t be the trace with Almost Same Transition check.

1. Set i to equal 0.
2. If i equals n+1 return t.
3. Correct the trace t using the algorithm in Figure 4.20 with A_i and M as input.
4. Increase i, then go to 2.

Figure 4.21 Counterexample Correction Algorithm
for Almost Same Transition Check

This algorithm is based on the algorithm in Figure 4.20 and needs to remember what new events replace what original events in what automata.

4.2.6 Evaluation

This test takes the heuristic minS for the second step of candidate selection and the projection limit is 3000. The non-projection model checker used for the checking is the BDD Language Inclusion Checker. The examples used in this evaluation are from the PROIsafe field bus protocol [MM03], [MM02], [PN02].

Model	Property		Without Automata Simplification			With Automata Simplification		
			Nodes	Time(s)	O	Nodes	Time(s)	O
profisafe_i4_host	HOST__fv_crc__property	F	1	30.915	4	1	320.586	0
	HOST__fv_crc_noinit__property	T	482583	51.816	2	1	550.825	0
	HOST__fv_timeout__property	T	1	26.063	4	1	7.333	0
profisafe_i4_slave	SLAVE__fv__property	T	1	1.174	0	1	1.431	0
profisafe_i4	HOST__fv_crc__property	F	200701	70.888	13	127134	260.39	11
	HOST__fv_crc_noinit__property	F	213800	74.151	13	125906	313.274	11
profisafe_o4	HOST__fv_crc__property	F		O	18	1336962	575.188	18
	HOST__fv_crc_noinit__property	F		O	21	769108	2037.814	16
profisafe_o4_host	HOST__fv_crc__property	F	1	27.312	0	1	295.411	0
	HOST__fv_crc_noinit__property	T	1	60.223	2	1	497.484	0
	HOST__fv_timeout__property	T	1	14.95	1	1	7.908	0
profisafe_o4_slave	SLAVE__fv__property	T	155472	19.571	4	1	62.745	0

Table 4.3: Language Inclusion Check Results
with and without Automata Simplification

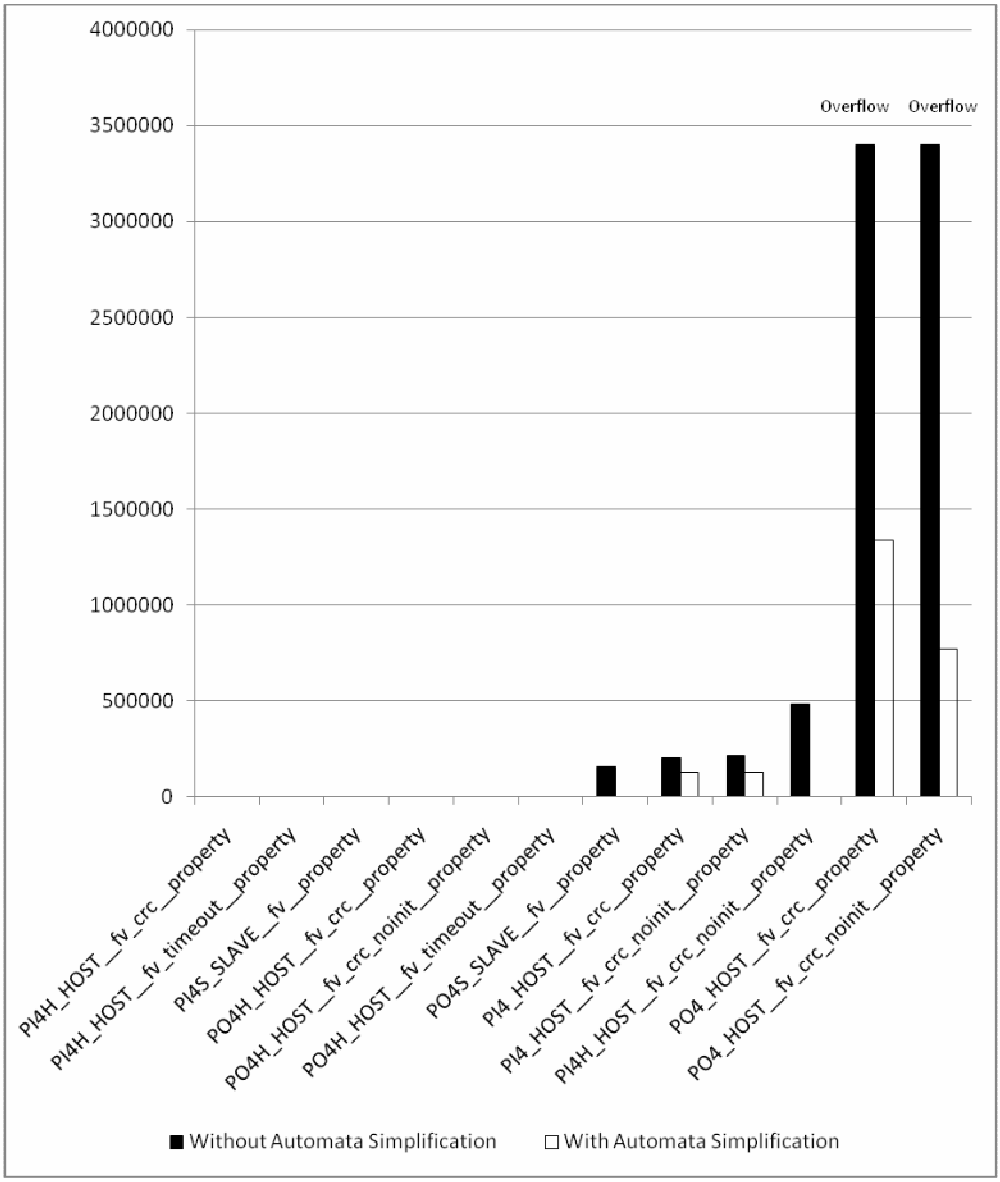


Figure 4.22: Peak Nodes after projection

with and without Automata Simpilification

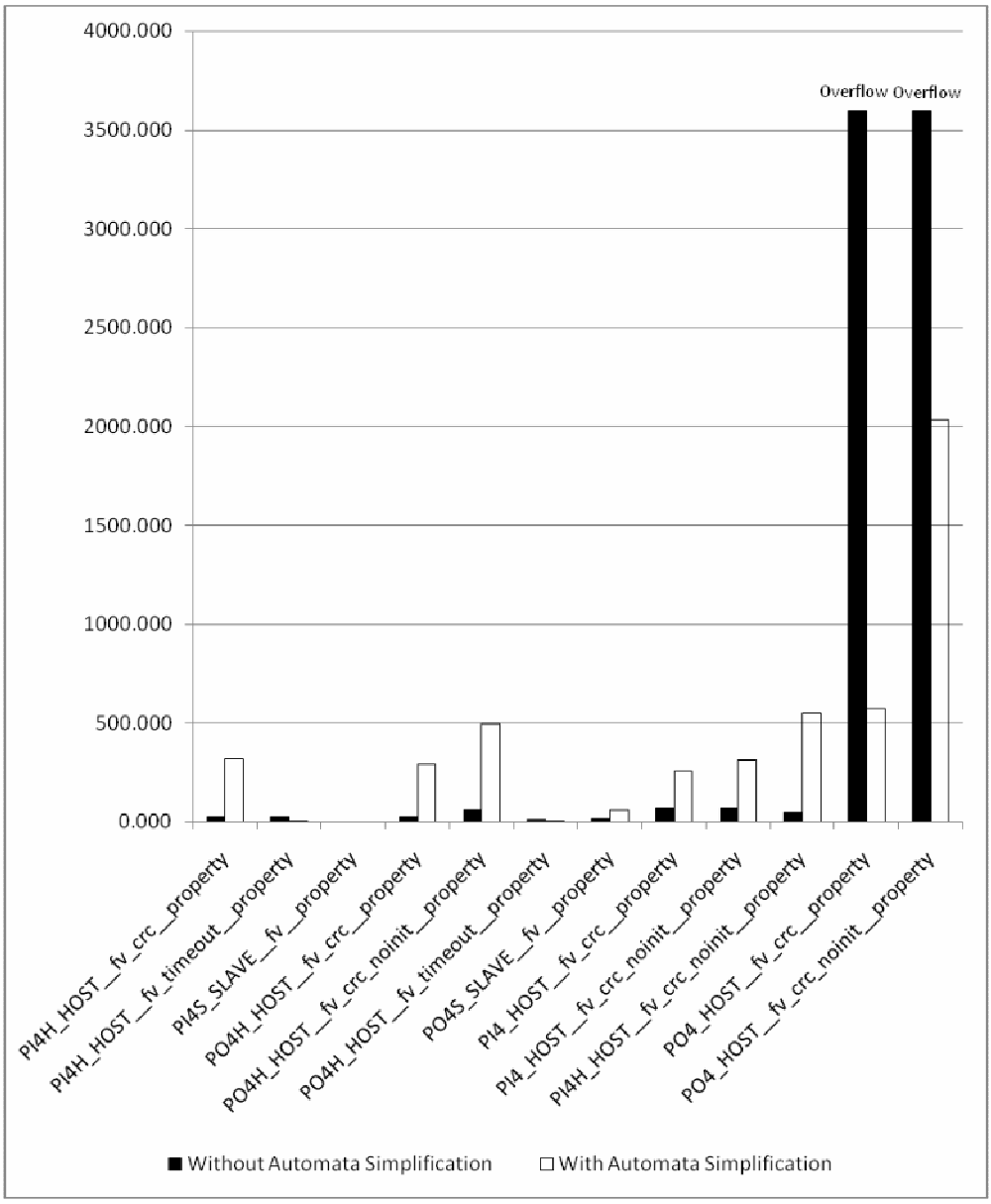


Figure 4.23: Language Inclusion Check Process Time

with and without Automata Simplification

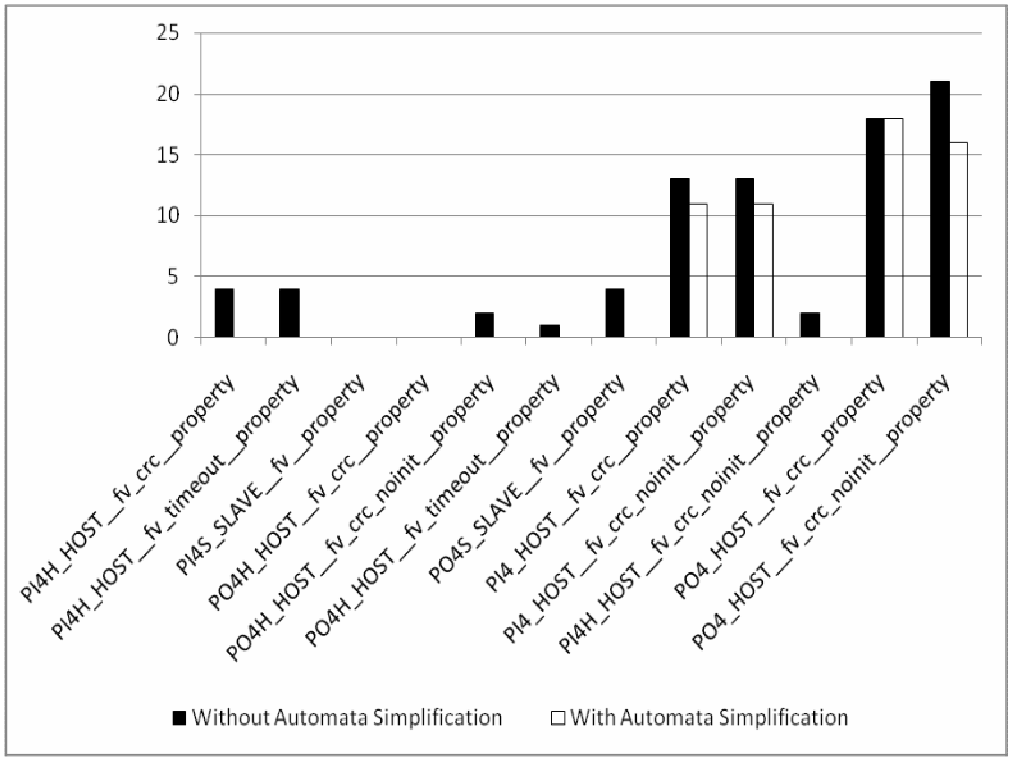


Figure 4.24: Projection Overflows with and without Automata Simplification

Table 4.3 displays two test cases. The first case is to check the model without Automata Simplification. The second case is to check the model with Automata Simplification. The content of the test includes three parts which are Nodes, Time, and O. The Nodes is the peak number of the nodes of the synchronous product of the model after projection that the BDD language inclusion checker explores. The Time is the processing timefor the model checking. The O states the overflows of projection. Furthermore, there is only one property in each example.

According to Table 4.3, the peak number of nodes explored by the BDD checker after projection with Automata Simplification is smaller than the peak number of nodes after projection without Automata Simplification. In Figure 4.22, it shows that result clearly. Furthermore, there are two examples which are not solved without Automata Simplification, but solved with Automata Simplification. There

two examples are property *HOST__fv_crc__property* and property *HOST__fv_crc_noinit__property* in model *profisafe_o4* [MM03], [MM02], [PN02]. The model checking processing timewith Automata Simplification is slower than the processing timewithout Automata Simplification, except for the models which cannot be solved without Automata Simplification. After Automata Simplification, the model becomes small enough for the compositional model checker to do more projection with the same projection limit. With Automata Simplification, the model is smaller and the number of projection overflows is smaller, too. According to Figure 4.24, no model with Automata Simplification has more overflows than the model without Automata Simplification.

Model	Property	Without AS	With AS		
		P Time(s)	P Time(s)	S Time(s)	W Time(s)
profisafe_i4_host	HOST__fv_crc__property	26.304	5.873	279.148	320.586
	HOST__fv_crc_noinit__property	19.013	8.466	513.358	550.825
	HOST__fv_timeout__property	22.494	1.302	5.072	7.333
profisafe_i4_slave	SLAVE__fv__property	0.740	0.474	0.580	1.431
profisafe_i4	HOST__fv_crc__property	22.412	6.979	174.293	260.390
	HOST__fv_crc_noinit__property	24.923	6.002	205.019	313.274
profisafe_o4	HOST__fv_crc__property	25.085	13.607	390.413	575.188
	HOST__fv_crc_noinit__property	15.458	13.695	1838.376	2037.814
profisafe_o4_host	HOST__fv_crc__property	26.695	5.677	285.195	295.411
	HOST__fv_crc_noinit__property	56.612	8.469	480.952	497.484
	HOST__fv_timeout__property	12.643	1.389	4.669	7.908
profisafe_o4_slave	SLAVE__fv__property	8.118	2.570	47.259	62.745

Table 4.4: Projection Time and Almost Same Transition Check Time

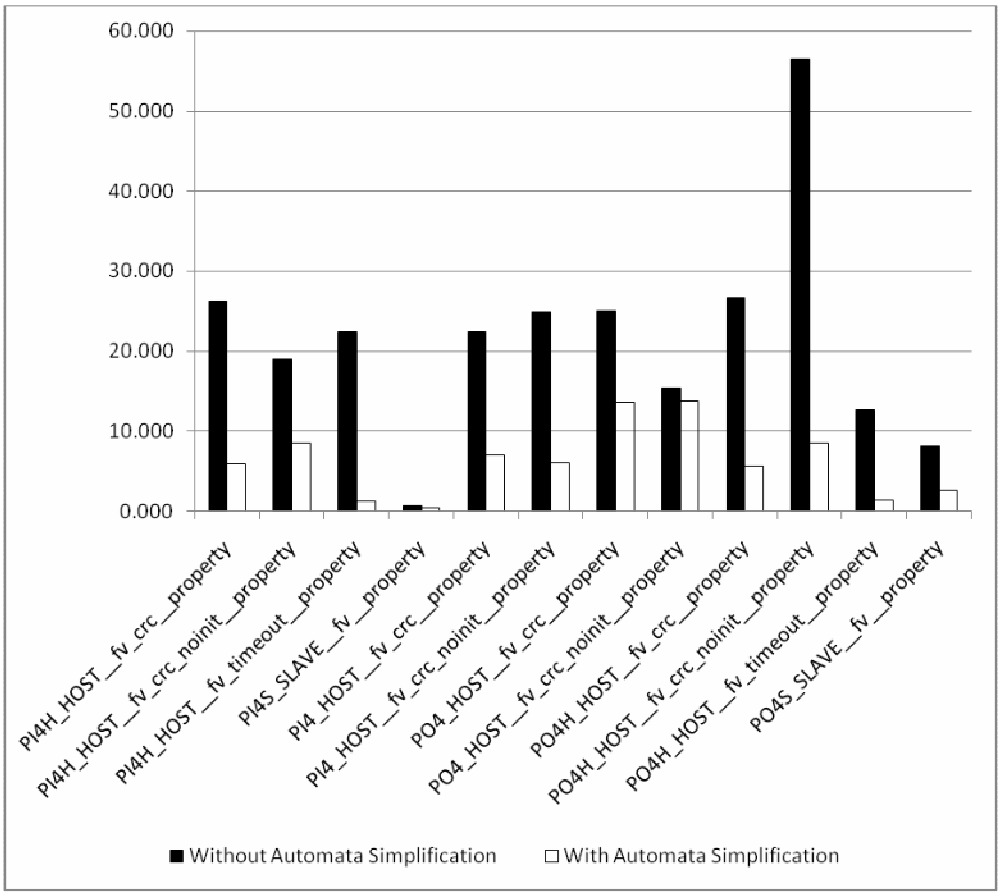


Figure 4.25: Projection Time with and without Automata Simplification

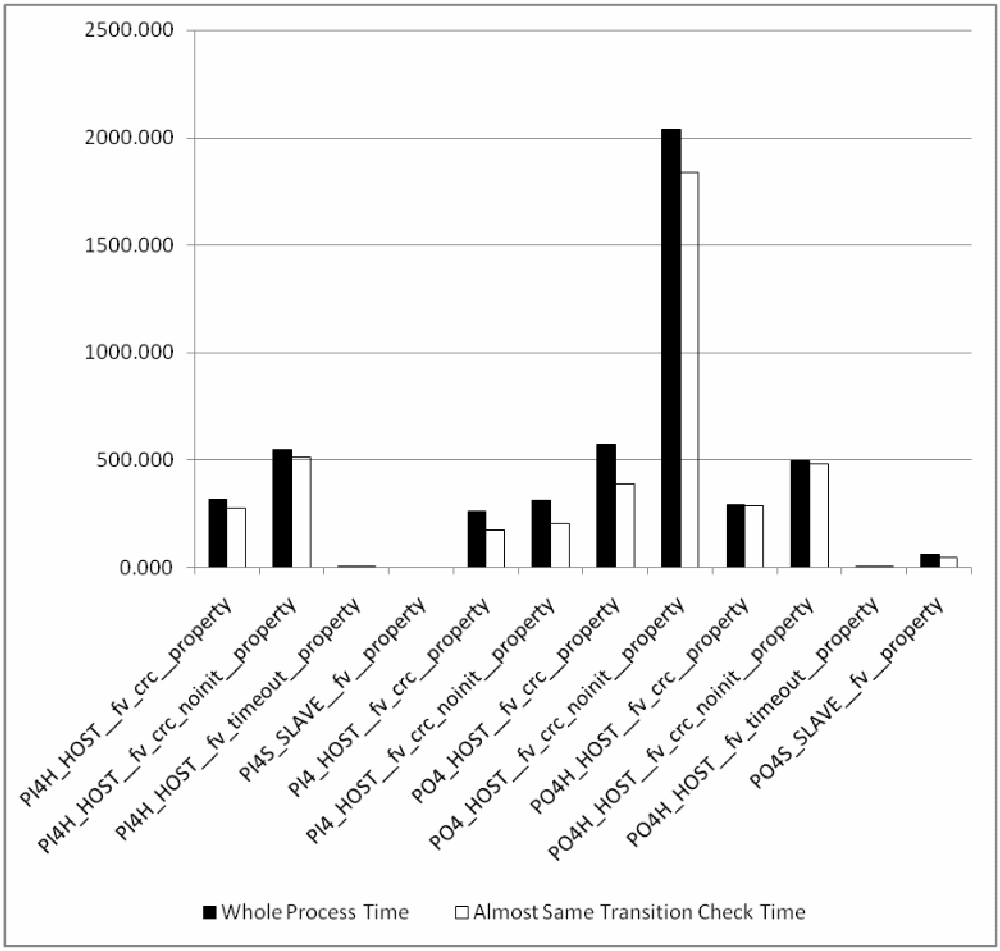


Figure 4.26: Whole Processing timeand Almost Same Transition Check Time

Table 4.4 describes the projection time with and without Automata Simplification, the Almost Same Transition Check time, and the whole processing time with Automata Simplification. In Table 4.4, P time represents projection time; S time represents Almost Same Transition Check time; W time represents the whole processing time of model checking with Automata Simplification. According to Table 4.4 and Figure 4.25, the projection time with Automata Simplification is longer than the projection time with Automata Simplification. The reason is that the automata of the model are smaller after Automata Simplification and that makes projection faster even though there are more projections after Automata

Simplification. However, Almost Same Transition Check is very time consuming, which can be seen from Table 4.4 and Figure 4.26, too. According to Figure 4.26, the Almost Same Transition Check time occupies nearly or over ninety percent of the whole model checking processing time. Therefore, Almost Same Transition Check is the most time consuming check in Automata Simplification, while selfloop check and same transition check are not very time consuming.

As a result, the Automata Simplification strategy can reduce the size of the synchronous product of the model and improve the performance of verification for large examples. Furthermore, some examples can only be solved with Automata Simplification. However, this is a time consuming strategy.

Chapter 5

Candidates Construction and Selection

The approach of the compositional verification in this report is constructing a candidate list, choosing a candidate from it, and applying projection. If the automata are composed in different order, the performance of verification is different. This chapter presents the method mustL [FM06] to construct a candidate list, and four heuristics which are maxL, minS, minT, [FM06] and minCut to pick a candidate from the candidate list. MustL is explained first and revisited later for its enhancement. The four heuristics are used to explore how the different order influences the different performance. Also, this chapter gives some evaluation for these heuristics.

5.1 Candidates Construction

5.1.1 Introduction

A key issue in compositional verification is in which order and what automata should be composed. Before choosing a proper set of automata for composing, it is necessary to construct a set of candidates to be chosen from. There is more than one automaton in a model usually and these automata can be combined variously. Therefore, constructing a proper set of candidates is an important part in compositional verification. This set of candidates needs to be small enough and has good value in use of compositional verification. MustL [FM06] is such a method to construct a series of candidates for automata composing and projection.

5.1.2 MustL candidates construction algorithm

In a candidate, there are two factors should be considered in this algorithm. One of them is the set of automata which forms the candidate. And the other factor is the set of hidden events which is local in this candidate and can be projected. Candidates are “equal” if their sets of automata are equal no matter whether their sets of hidden events are equal. If their sets of hidden events are not equal, then integrate the sets of hidden events into one as the set of hidden events of the candidates.

Let P be the plant automata set of the model and C be the candidate list. Let E be the events set which contains all events mentioned by P only.

1. Initialize candidate list C to be empty. Let $E = \{e_1, \ldots e_n\}$ and $i = 1$.
2. Find the set of automata using e_i in P and form it into a candidate *can* with the hidden event e_i.
3. If *can* is a new candidate, then go to 4. Else go to 6.
4. If *can* is a subset of some existent candidates in C, then find the existent candidates and add e_i into each of them as a hidden event. If *can* includes some existent candidates in C, then set the hidden events of *can* to be the union of the sets of hidden events of the existent candidates plus event e_i.
5. Add *can* to C. Go to 7.
6. If *can* is not a new candidate, then find the existent candidate and the candidates which subsume *can* in C and add the new hidden event e_i.
7. Let $i = i + 1$, if $i \leq n$, go to 2. Else go to 8.
8. Return candidate list C.

Figure 5.1: MustL candidates construction algorithm

Figure 5.1 describes the mustL candidates construction algorithm. In a model, for each event, there exists a candidate which is the set of automata using that particular event; therefore it is possible to find out all of the different combinations of automata shared some local events with this method. These different combinations are candidates for further candidate selection. Also, this method is based on the local events, and compositional verification uses projection to improve the performance of verification. Thus this method is

suitable for projection which is an effective way to simplify a model by removing some local events. Generally, this algorithm picks some sets of automata which share some local events from the model and forms them into a candidate list for the next step.

5.2 Heuristic MaxL

5.2.1 Introduction

Heuristic maxL is a method used to pick a candidate with the maximum proportion of local events from a list of candidates [FM06].

This project uses projection as a way to simplify automata. The major idea of projection is to remove certain events, which are local events, in order to simplify an automaton. Therefore, if the local events occupy a large proportion of all events of the automaton, the automaton after projection is likely to be much smaller than the original automaton. A candidate normally contains more than one automaton. The automata in a candidate will be composed into one automaton and simplified. Therefore, if the local events occupy larger proportion of all events of the automata in the candidate, the automaton after composing and projection is likely to be much smaller too. That is to say, if the candidate has the maximum proportion of local events, the new automaton of this candidate is likely to be the furthest simplified automaton in all candidates.

Consequently, heuristic maxL is capable of candidate selection.

5.2.2 Algorithm for calculating proportion of local events

A candidate has two properties which are local events and all events. Local events are events shared by the automata in the candidate only. All events are all of events used by the automata in the candidate. Set the size of local events with *L* and set the size of all events with *E*. Then the proportion of local events is

L / E

For example, in a candidate, the size of local events is 6 and the size of all events is 20. Then the proportion of local events is

6 / 20 = 30%

5.3 Heuristic MinS

5.3.1 Introduction

Heuristic minS is a method used to select the candidate with the minimum number of states of the synchronous product [FM06]. This heuristic tries to keep the synchronous product of the candidate after composing and projection as small as possible.

Model verifiers prefer small and simple models. The smaller the model is, the faster the verifier can check the model. The aim of composing and projection is to reduce the size of large models and enable the verifier to verify them. Hence, if the synchronous product of a candidate is not small enough, this candidate is not considered to be composed and projected. Heuristic minS is such a heuristic to choose the candidate with the smallest number of states of the synchronous product.

However, it is unwise to calculate the actual and precise number of states, because constructing the synchronous product is the only way to obtain the actual and precise number of states, while synchronous product constructing is a time and space consuming procedure, especially for large models. Therefore, for heuristic minS, it is better to predict the number of states of the synchronous product rather than calculate the real number of states.

5.3.2 Algorithm for predicting the number of states of the synchronous product

There are many algorithms for predicting the number of states of the synchronous product. For example, machine learning techniques [PS08] can be used for the prediction and the machine learning way produces a more accurate result also. Machine learning has its drawbacks. It costs too much time and memory to train the data. This project takes a much more ordinary algorithm to predict the number of states of the synchronous product of a candidate in order to validate the feasibility of heuristic minS.

This algorithm assumes that the number of states of the synchronous product of a candidate is the product of multiplying every number of states of the automata in the candidate, if these automata are composed without projection. With projection, the states number of the synchronous product is reduced in proportion to the amount of local events in all events. For automata $A_1, \ldots A_n$ with number of states $P_1, \ldots P_n$, set the number of local events to L and set the number of all events to E. Then the estimated number of states of the synchronous product of the candidate is

$$\mathbf{P_1 \times \ldots \times P_n \times (E - L) / E}$$

For example, there are three automata which are automaton *A*, automaton *B* and automaton *C* in a candidate. The numbers of states of *A*, *B* and *C* are 3, 5, and 4 respectively. The number of local events is 6 and the number of all events is 20. Then the estimated number of states of the synchronous product of this candidate is

$$(3 \times 5 \times 4) \times (20 - 6) / 20 = 42$$

5.3.3 Number of states prediction evaluation

Model	Candidate	Predicted state number	Real state number
bfactory	machine3	0.75	1
	machine1:plant	1.5	2
	machine2:plant	1.5	2
Profisafe_i4_slave SLAVE__fv__property	SLAVE__out_ps_status_bit4_FV__app_process_data__INPUT:plant SLAVE__uncont:plant	7.8	7
	SLAVE__got_msg SLAVE__main SLAVE__polling	35.57	17
	SLAVE__fv__property:plant SLAVE__polling SLAVE__uncont:plant	51.55	16
	SLAVE__fv__property:plant SLAVE__got_slave_timeout:plant SLAVE__slave_timeout:plant SLAVE__uncont:plant	102.92	290
	SLAVE__fv__property:plant SLAVE__got_msg SLAVE__in_CRC SLAVE__in_cons_num__4 SLAVE__uncont:plant	853.89	1325
	SLAVE__in_CRC SLAVE__in_cons_num__4 SLAVE__main SLAVE__out_cons_num_spec__4 SLAVE__polling	1523.57	489
	SLAVE__got_slave_timeout:plant SLAVE__main SLAVE__number_ok_cycles__INPUT SLAVE__out_ps_status_bit3_TO:plant SLAVE__polling SLAVE__slave_timeout:plant	1711.38	47
	SLAVE__got_slave_timeout:plant SLAVE__in_cons_num__4 SLAVE__main SLAVE__number_ok_cycles__INPUT SLAVE__out_cons_num_plant__4:plant SLAVE__out_cons_num_spec__4 SLAVE__out_ps_status_bit2_CRCNO:plant SLAVE__polling SLAVE__slave_timeout:plant	206526.31	O

Table 5.1: Predicted state number and real state number for controllability check and language inclusion check

In order to evaluate the states number prediction algorithm, this section uses two candidate lists of two models to do the evaluation. The first model is *bfactory* with

a candidate list which has three candidates. This model is verified for controllability. The second model is *Profisafe_i4_slave* with property *SLAVE__fv__property* and a candidate list which has eight candidates. This model is verified for language inclusion. The state limit for projection is 1000. Table 5.1 describes the candidates and the automata in it, and the predicted state number and real state number of the synchronous product of these candidates.

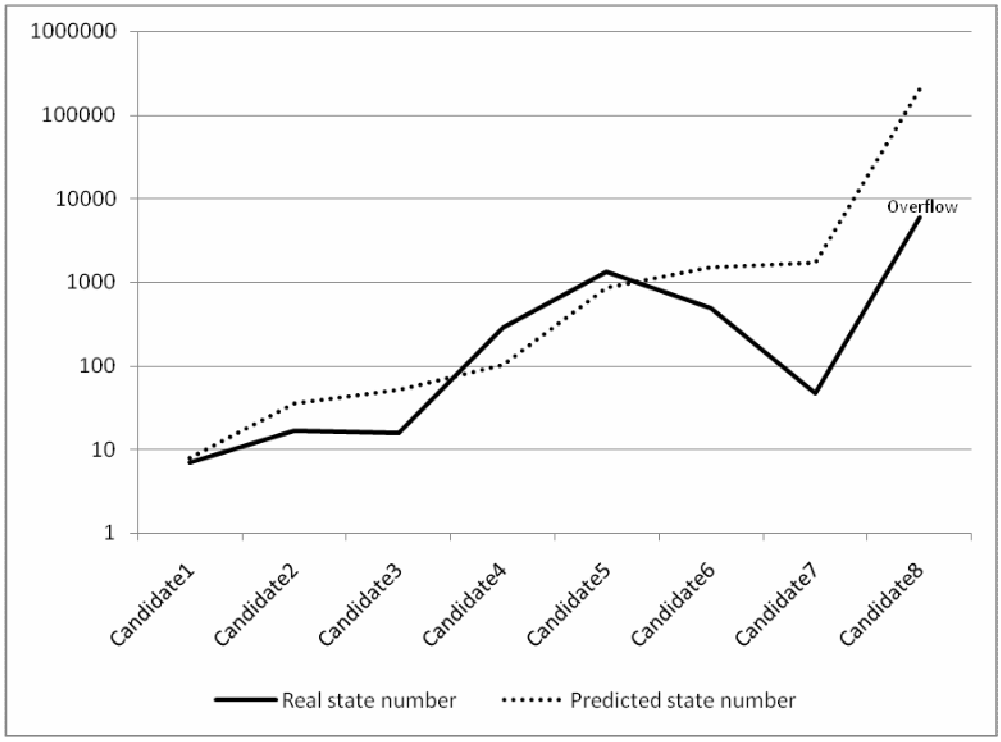


Figure 5.2: Profisafe_i4_slave predicted states nubmer and real states number for language inclusion check

According to Table 5.1, the predicted state number is different from the real state number. The projection for the eighth candidate of model *profisafe_i4_slave* is overflow, and the real number of states of it cannot be calculated. However, according to Figure 5.2, the trend of the increment of the predicted state number is

similar to the trend of increment of the real state number. This is enough for candidate selection because this project only considers the candidate with the minimum number of states of the synchronous product, and not the real and accurate number of states. Since this algorithm still cannot completely represent the proper trend at some places, some improvement may be possible to enhance the prediction.

5.4 Heuristic MinT

5.4.1 Introduction

Heuristic minT is a method used to choose the candidate with the minimum number of transitions of the synchronous product [FM06].

This heuristic focuses on the structure of automata, too. Furthermore, heuristic minT is keen to keep the synchronous product of the candidate after composing and projection as small as possible like heuristic minS. Instead of using the number of states as the measure of complexity of an automaton, this heuristic uses the number of transitions. The number of states is not the only factor to evaluate if an automaton is small or not, because there is another factor needs to be considered which is number of transitions. For example, if an automaton has two states, but it has two hundred transitions, this automaton is not a small automaton.

Accordingly, heuristic minT is also a strategy which is capable of selecting the candidate with a small synchronous product.

5.4.2 Algorithm for predicting the number of transitions of the synchronous product

The number of transitions of the synchronous product of a candidate cannot be calculated directly because of the same reason of calculating the number of states of the synchronous product. Therefore, finding a method to predict the number of transitions of the synchronous product is necessary for heuristic minT.

This algorithm only considers the transitions labeled with events which are not local events. For each event which is not a local event, find all the transitions labeled with it in every automaton in the candidate. Multiply the number of these transitions of every automaton and obtain the product. Then add the entire product of all events except local events together and obtain a sum. This algorithm assumes the sum as the predicted number of transitions of the synchronous product of the candidate. For automata $A_1, \ldots A_n$ with number of σ transitions $T_{1,\sigma}, \ldots T_{n,\sigma}$, set the non-local events set to *NE*. Then the estimated number of transitions of the synchronous product of the candidate is

$$\sum_{\sigma \in NE} \prod_{i=1}^{n} T_{i,\sigma}$$

For example, there is a candidate with three automata which are automaton *A*, automaton *B*, and automaton *C*. This candidate has three events which are event *a*, event *b*, and event *c*. These events are not local. The numbers of transitions labeled with event *a* are 1 in automaton *A*, 4 in automaton *B*, and 7 in automaton

C. The numbers of transitions labeled with event *b* are 2, 5, and 8 in automaton *A*, *B*, and *C* respectively. The numbers of transitions of event *c* are 3, 6, and 9 respectively. Then the predicted transitions number of the synchronous product of the candidate is

$$(1 \times 4 \times 7) + (2 \times 5 \times 8) + (3 \times 6 \times 9) = 28 + 80 + 162 = 270$$

5.4.3 Number of transitions prediction evaluation

This section uses two candidate lists of two models in order to evaluate the number of transitions prediction algorithm. The first model is *bfactory* with a candidate list which has three candidates. This model is verified for controllability. The second model is *Profisafe_i4_slave* with property *SLAVE__fv__property* and a candidate list which has eight candidates. This model is verified for language inclusion. The node limit for projection is 1000.

Model	Candidate	Predicted transition number	Real transition number
bfactory	machine3	1	1
	machine1:plant	3	4
	machine2:plant	3	4
Profisafe_i4_slave SLAVE__fv__property	SLAVE__got_msg SLAVE__main SLAVE__polling	1716	253
	SLAVE__out_ps_status_bit4_FV__app_process_data__INPUT:plant SLAVE__uncont:plant	1732	345
	SLAVE__fv__property:plant SLAVE__got_slave_timeout:plant SLAVE__slave_timeout:plant SLAVE__uncont:plant	53752	23274
	SLAVE__fv__property:plant SLAVE__polling SLAVE__uncont:plant	113889	1564
	SLAVE__fv__property:plant SLAVE__got_msg SLAVE__in_CRC SLAVE__in_cons_num__4 SLAVE__uncont:plant	134094	80936
	SLAVE__in_CRC SLAVE__in_cons_num__4 SLAVE__polling SLAVE__main SLAVE__out_cons_num_spec__4	3655286	5669
	SLAVE__got_slave_timeout:plant SLAVE__main SLAVE__number_ok_cycles__INPUT SLAVE__out_ps_status_bit3_TO:plant SLAVE__polling SLAVE__slave_timeout:plant	2.15E8	1103
	SLAVE__got_slave_timeout:plant SLAVE__in_cons_num__4 SLAVE__main SLAVE__number_ok_cycles__INPUT SLAVE__out_cons_num_plant__4:plant SLAVE__out_cons_num_spec__4 SLAVE__out_ps_status_bit2_CRCNO:plant SLAVE__polling SLAVE__slave_timeout:plant	5.92E13	O

Table 5.2: Predicted transition number and real transition number for controllability check and language inclusion check

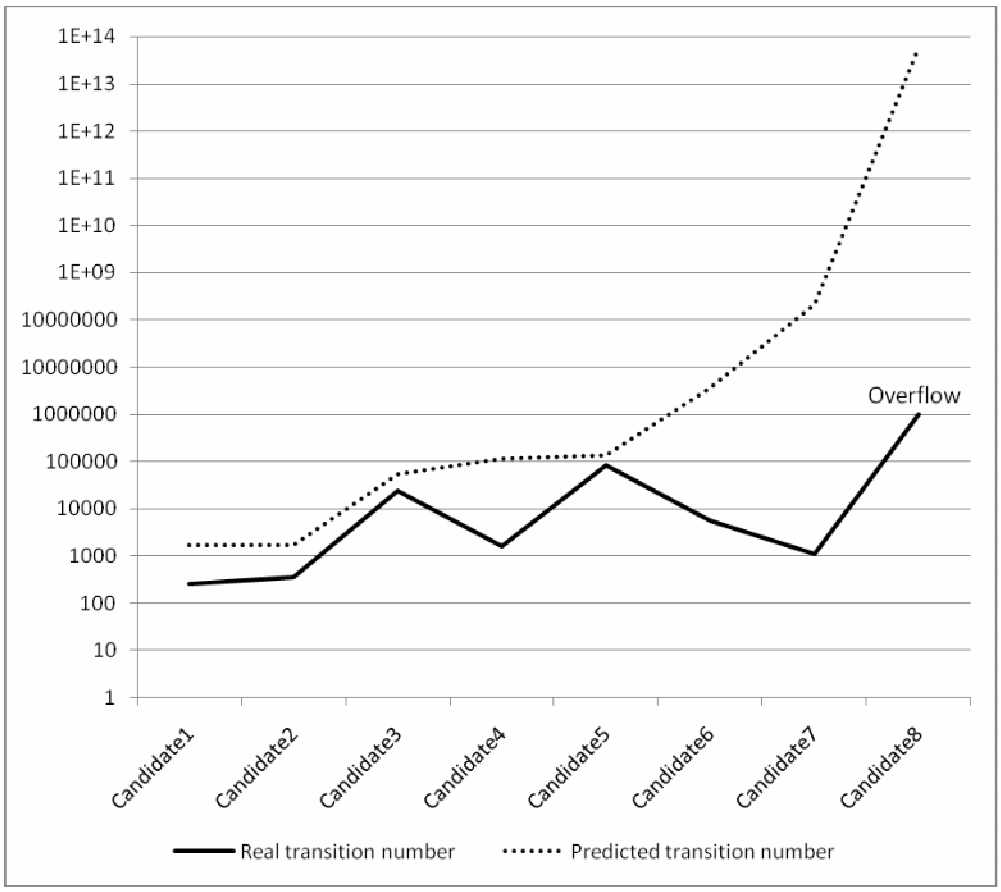


Figure 5.3: Profisafe_i4_slave predicted transition nubmer
and real transition number for language inclusion check

According to Table 5.2, the predicted number of transitions is different from the real number of transitions. The projection for the eighth candidate of model *profisafe_i4_slave* is overflow, and the real number of transitions of it cannot be calculated. However, according to Figure 5.3, the trend of the increment of the predicted number of transitions is similar to the trend of increment of the real transitions number. At least, the candidates with the minimum number of transitions are same. This is enough for candidate selection because this project only considers the candidate with the minimum transitions number of the synchronous product and not the real and accurate transitions number. Since this

algorithm still cannot completely represent the proper trend at some places, some improvement may be possible to enhance the prediction.

5.5 Heuristic MinCut

5.5.1 Introduction

The three heuristics above are all based on the structure of automata such as events, states, and transitions. Respectively, heuristic maxL concentrates on events; heuristic minS concentrates on states; and heuristic minT concentrates on transitions. Apart from that, according the study of modular structure of those models, a quite different new heuristic minCut is introduced in this project. This heuristic is based on the connections between automata instead of the structure of single automaton.

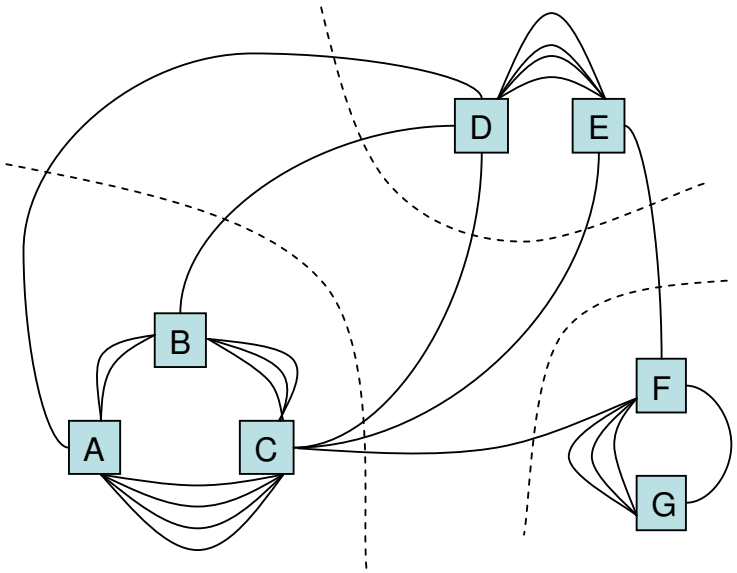


Figure 5.4: Simple modular structure graph

Figure 5.4 describes a modular structure of a model. In the graph of Figure 5.4, every letter represents an automaton of the model, and the line between them represents that they share an event. More lines mean more events are shared. In the structure graph, the automata with more shared events are grouped together and therefore the model is partitioned into several groups. For example in Figure 5.4, there are seven automata *A*, *B*, *C*, *D*, *E*, *F*, and *G*. In these automata, *A*, *B*, and *C* are close to each other and grouped together. In this way, there are two more groups which are *D* and *E*, and *F* and *G*. Therefore, if the automata in one group are composed first, the synchronous product of this group could be simplified much further, and more local events can be projected out. The reason is in one group, the automata share much more events with each other than the rest of the groups. Also, if the group has fewer connections with outside, then the synchronous product of the group could be smaller. For example, if the group has no connections with other groups, the group could be eliminated from the model entirely. Hence, finding the group with the fewest connections with outside and composing its automata first is another possible approach to improve the performance of composing and projection.

In order to find such a group according to the modular structure, a new strategy based on graph theory is introduced. Just as its name implies, graph theory is the study of graphs [GC85]. As for this project, the graphs are modular structure graphs of models. The modular structure graph shows the relations between automata by events they share. In graph theory, a theorem named max-flow min-cut is used to determine the best way to cut a graph. It states that “the maximum amount of flow is equal to the capacity of a minimum cut.” [MFMC] However, this method needs a source and a sink (target) for calculating the flows, while the modular structure graph is an undirected graph and does not have a

source state and a sink state. Fortunately, the idea of adopting max-flow min-cut theorem is to find the minimum cut and the cut number can be calculated without knowing the amount of flow. Thus, a reformative max-flow min-cut method named “minCut” is introduced to select the candidate.

MinCut is a heuristic to select the candidate with the minimum cut number. It comes from the max-flow min-cut theorem and focus on the connections of automata. This heuristic does not calculate the cut number by the flow like the max-flow min-cut theorem does. It obtains the cut number by counting the shared events instead. This heuristic divides the model into two groups. One group is the candidate to be composed and the rest of the model is another group. The number of events shared by the two groups is the cut number.

5.5.2 Algorithm for calculating the cut number

The way of calculating the cut number is picking a group out, which is taken as the source, taking the rest of the automata in the model as the sink, and then adding every shared event between every automaton in the source and every automaton in the sink. The amount of shared events is the cut number. For example,

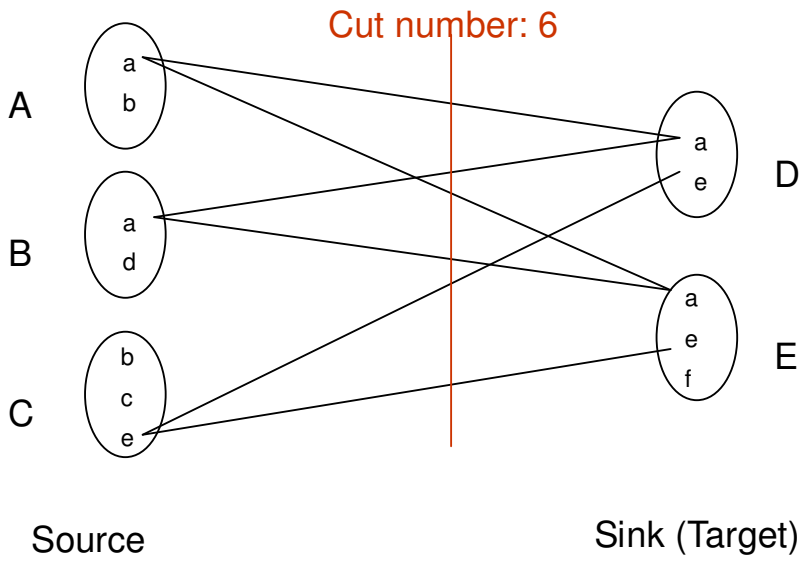


Figure 5.5: A sample of cut number calculation

This model in Figure 5.5 has five automata which are *A*, *B*, *C*, *D*, and *E*. In which, *A*, *B*, and *C* are grouped together and taken as source and the rest automata which are *D* and *E* are taken as sink. So for every automaton in source, every event is counted once for each time it is shared with an automaton in sink. Furthermore, if this event is shared more than once, it should be counted more than once. In the example in Figure 5.5, the event *a* in automaton *A* is shared by automaton *D* and *E*, so event *a* is counted twice with respect to automaton *A*. So is the event *a* in automaton *B*. In this way, the cut number for group *ABC* is six.

One of the disadvantages of heuristic minCut is that it is hard to form the groups. It will grow exponentially with the number of automata in the model if all of the combinations of the automata are considered. Accordingly, this heuristic cannot be used alone for candidate selection of large systems. Nevertheless, in this project,

some candidates have been constructed already at the first step and they can be treated as the sources. In this way, minCut can be adopted to select the candidate.

5.6 An Enhancement for Candidates Construction

5.6.1 Subsumption

The first section of this chapter describes a candidates construction method musL. This method constructs a candidate list. However, this candidate list may not be small enough. There may be some candidates which are subsets of other candidates in the candidate list. If the compositional verification composes the larger candidates first, it is slower than composing the subset of those larger candidates first. That means this method only needs to consider the smaller candidates and abandon the larger candidates which contain these smaller candidates. The subsumption test is to check whether a candidate subsumes another candidate. The benefits of subsumption test are making the candidate list short and improving the candidate selection speed by removing those larger candidates from the candidate list. Therefore, some improvements can be made using subsumption test in the algorithm in Figure 5.1.

Let P be the plant automata set of the model and C be the candidate list. Let E be the events set which contains all events mentioned by P only.

1. Initialize candidate list C to be empty.
2. Let $E = \{e_1, \ldots e_n\}$ and $i = 1$, find the set of automata using e_i in P and form it into a candidate *can* with the hidden event e_i.
3. If *can* is a new candidate, then go to 4 and check subsumption. Else go to 6.
4. If *can* is a subset of some existent candidates in C, then replace these existent candidates with *can* and go to 7. If can includes some existent candidates in C, then drop *can* and go to 7.
5. Add *can* into C.
6. If can is not a new candidate, then find the existent candidate in C and add the new hidden event e_i.
7. Let $i = i + 1$, if $i \leq n$, go to 2. Else go to 8.
8. Return candidate list C.

Figure 5.6: MustL candidates construction algorithm with subsumption test

According to the algorithm in Figure 5.6, only small candidates can remain in C. That is to say, if the new candidate is a subset of an existent candidate then replace the existent candidate with the new candidate in C or if there is an existent candidate which is a subset of the new candidate then drop the new candidate. In this way, candidates construction with subsumption test can minimize the size of C and speed up the candidate selection. Further, candidates construction with subsumption test can accelerate compositional verification.

5.6.2 Subsumption evaluation

This test uses mustL as the first step and the maxL as the second step and does the controllability test of the examples. The states limit for projection is 1000. The non-projection model checker used in the test is the Modular model checker.

Model	without subsumption test		with subsumption test	
	States	Time (s)	States	Time (s)
big_bmw	0	1.063	0	1.417
ftechnik	1	13.199	1	11.343
fzelle	0	4.107	0	6.624
rhone_alps	0	0.413	0	3.274
rhone_tough	235256	15.6	5804890	39.273
tbed_ctct	697531	45.454	646329	18.06
tbed_nocoll	7798	59.67	8438	21.095
tbed_noderail	7464	55.535	8016	23.138
tbed_uncont	2550	53.033	2699	29.759
verriegel4	0	18.475	0	4.68

Table 5.3: Subsumption test with maxL

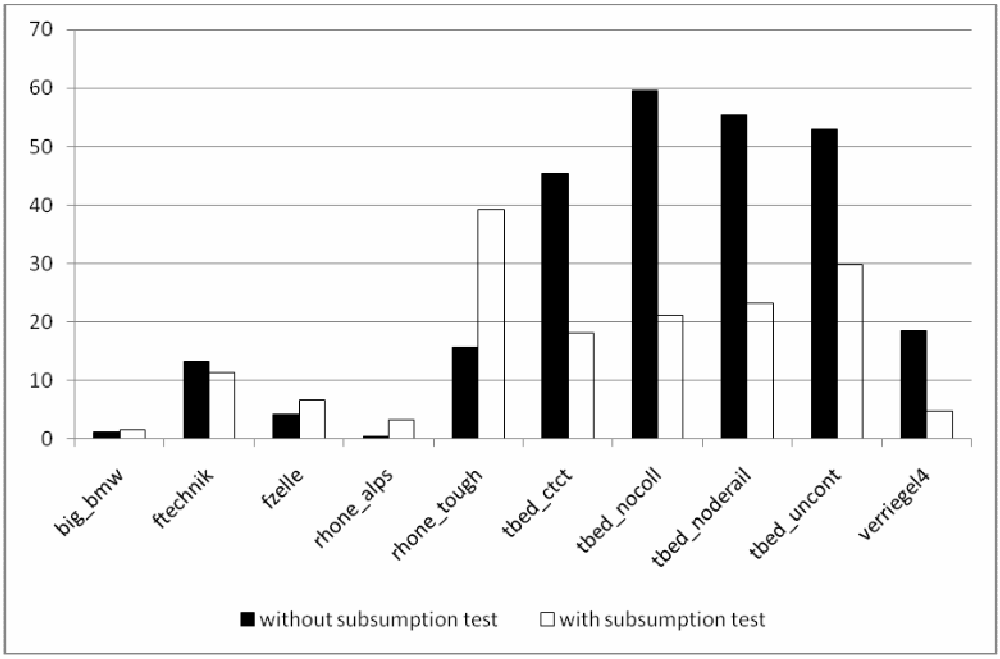


Figure 5.7: MaxL Processing Time with and without subsumption test

This test considers two parts for evaluation. One of parts is States and the other part is Time. States is the peak number of states encountered by the modular controllability checker when analyzing the simplified model. Time is the whole processing time of verification. The data in Table 5.3 shows that verification with subsumption test is faster than verification without it in most cases. But for some cases, especially for *rhone_rough*, it is slower. The reason is that subsumption test changes the order of composing with heuristic maxL too much and some improper candidates are selected. For example, there is a candidate list with two candidates C_1 and C_2. C_1 subsumes C_2. The model checker takes C_1 for projection without subsumption test. However, with subsumption test, C_1 is removed from the candidate list and the model checker can only takes C_2 for projection. Since the model checker takes different candidates when subsumption test is employed, the processing time is different. If subsumption test does not change the order of composing too much and only reduces the size of candidate list, the whole processing time can be cut down. Otherwise, the whole processing time may not

be cut down.

Since subsumption retains small candidates and removes large candidates, it is similar to heuristic minS. Both of these two methods prefer small candidates. Therefore, the subsumption will not change the order of composing for heuristic minS. The following test evaluates verification with heuristic minS and subsumption test.

The second test takes mustL as the first step and minS as the second step and does the controllability test of the examples. The states limit for projection is 1000. The non-projection model checker used in the test is the Modular model checker.

Model	without subsumption test		with subsumption test	
	States	Time (s)	States	Time (s)
big_bmw	0	0.506	0	0.482
ftechnik	1	6.223	2	3.534
fzelle	0	0.8	0	0.716
rhone_alps	0	0.236	0	0.224
rhone_tough	193297	3.907	193297	3.515
tbed_ctct	646329	10.485	646329	7.264
tbed_nocoll	7263	13.437	7263	8.393
tbed_noderail	7247	13.672	7247	7.718
tbed_uncont	2418	13.348	2418	7.947
verriegel4	0	2.304	0	2.021

Table 5.4: Subsumption test with minS

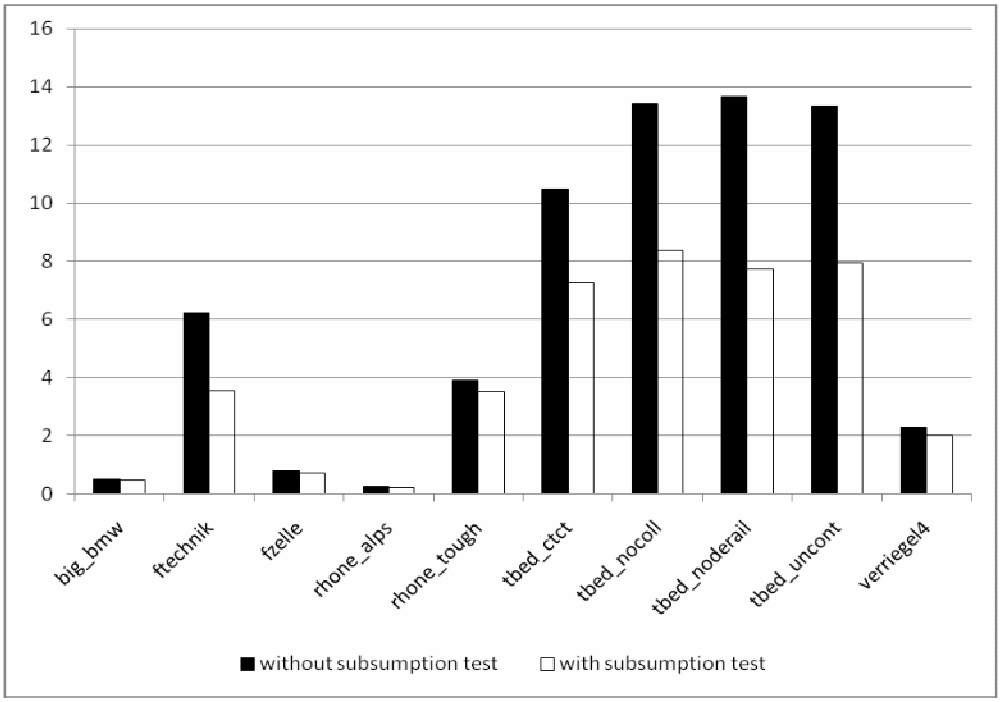


Figure 5.8: MinS Processing Time with and without subsumption test

This test considers two parts for evaluation. One of the parts is States and the other part is Time. States is the peak number of states encountered by the modular controllability checker when analyzing the simplified model. Time is the whole processing time of verification. According to the results in Table 5.4 and Figure 5.8, with subsumption test the verification time is almost cut down to half of the verification time without subsumption test. Meanwhile, it does not change the peak number of states in most of the examples because of the resemblance of minS and subsumption test which makes the subsumption test only reduce the number of candidates. The results in Table 5.2 prove that the subsumption test can really speed up the composing.

In order to explain why it can speed up processing, an example is considered for further analyzing. The example is *tbed_nocoll*, and its verification time is almost cut down to half of the original time after subsumption test is introduced.

step	without subsumption test	with subsumption test
1	105	58
2	104	59
3	103	59
4	102	60
5	101	59
6	100	60
7	99	60
8	98	59
9	97	60
10	96	60
11	95	60
12	94	59
13	91	58
14	90	57
15	89	58
16	88	59
17	87	60
18	86	61
19	85	61
20	84	62
21	81	61
22	80	60
23	79	59
24	78	59
25	77	59
26	76	57
27	75	56
28	74	55

step	without subsumption test	with subsumption test
29	73	54
30	72	53
31	71	52
32	70	51
33	69	51
34	68	51
35	67	51
36	66	49
37	65	49
38	64	45
39	63	42
40	62	38
41	61	37
42	60	38
43	59	38
44	58	39
45	57	38
46	55	36
47	52	35
48	50	36
49	49	36
50	47	33
51	46	30
52	44	29
53	43	26
54	41	21
55	39	21
56	37	19

Table 5.5: Candidates Number for every step

with and without subsumption test for tbed_nocoll

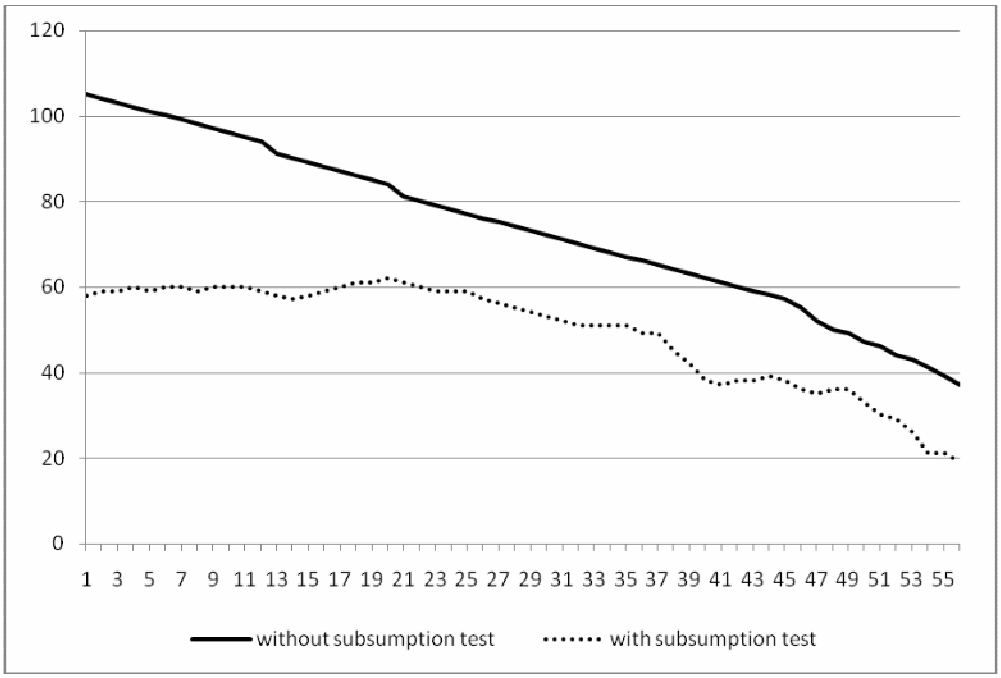


Figure 5.9: Candidates Number for every step
with and without subsumption test for tbed_nocoll

This example takes 56 steps to do the projection. That is to say, it chooses 56 candidates in total. At each step, there are some candidates to be chosen according to some method. Hence, if the candidates are too many, it will be more expensive to compare and choose the candidates. Table 5.5 shows the number of candidates in the candidate list for every step with and without subsumption test. It is clear from Table 5.5 and Figure 5.9 that at each step the candidate list with subsumption test has fewer candidates than the candidate list without subsumption test. The candidate list with subsumption test only has about half the number of candidates. That explains why the program with subsumption test is round about twice as fast.

5.7 Evaluation

5.7.1 Introduction

This project employs the results of controllability check and language inclusion check for the evaluation. Both controllability check and language inclusion check use four different heuristics, which are heuristic maxL, minS, minT and minCut, as the heuristic of the second step of candidate selection to check the model. As for the non-projection model checker, the state limit is set to ten million. That is to say, if the number of the states of the synchronous product of the model after composing and projection exceeds ten million, the checking will be terminated in order to prevent the non-projection model checker from running out of memory. Also, for the projection, there are two different state limits, which are three thousand and one thousand respectively. That means the number of the states of the new automaton, which is obtained by composing a set of automata and projecting the composition, cannot be larger than three thousand or one thousand. If the number exceeds the state limit, the composing and projection of this candidate stop. If the composing and projection of a candidate stop, this situation is called a projection overflow.

As for the controllability check, three aspects are examined. These three aspects are States, Time, and O. States represents the number of the states explored by the model checker. Time represents the processing time of the checking. O represents the number of the projection overflows. The non-projection controllability

checkers employed for the controllability check are Modular controllability checker, BDD controllability checker, and Native controllability checker. Nodes is examined instead of States for BDD controllability checker.

As for the BDD language inclusion check, four aspects are examined. These four aspects are States, Nodes, Time and O. States represents the sum of the number of states of the new automata created by the composing and projection. Nodes represents the peak number of the nodes explored by BDD model checker. Time represents the processing time of the checking. O represents the number of the projection overflows. This project takes only one property at a time to do the language inclusion check.

As for the Modular language inclusion check, three aspects are examined. These three aspects are States, Time and O. States represents the peak number of the states explored by Modular model checker. Time represents the processing time of the checking. O represents the number of the projection overflows. This project takes only one property at a time to do the language inclusion check.

5.7.2 Controllability Check Results

Table A1 and Table A2 in the appendix show the results of controllability check with projection limit three thousand and one thousand respectively for the Modular controllability checker. Data is summarized in Figure 5.10, 5.11 and 5.12.

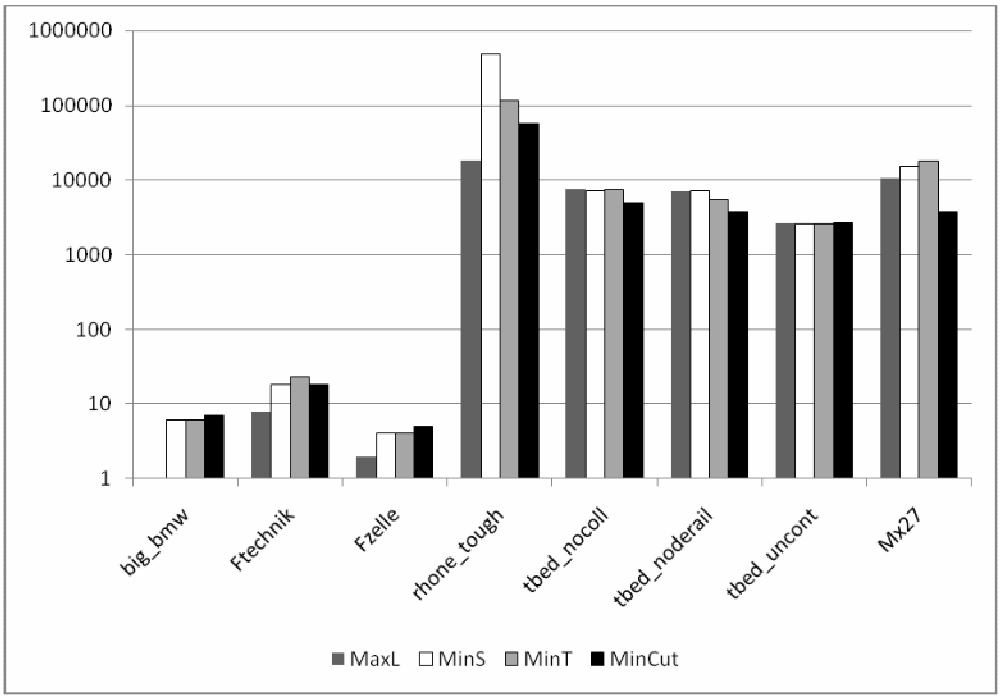


Figure 5.10: Peak number of states with Modular controllability checker
Projection limit 3000

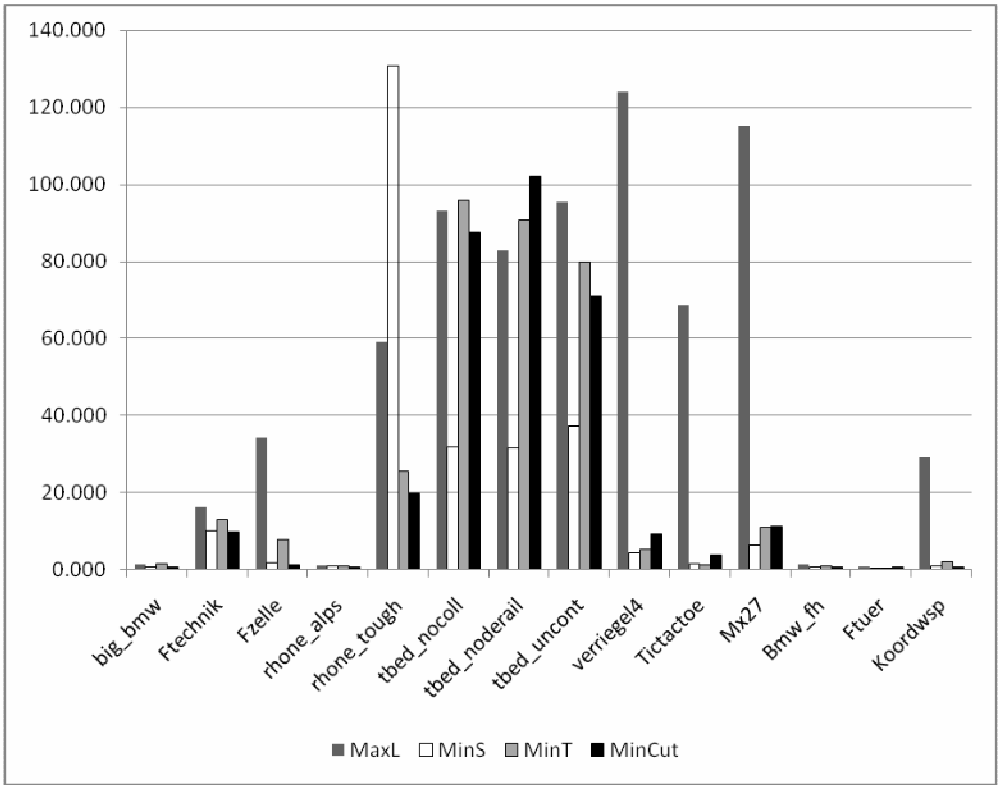


Figure 5.11: Processing time with Modular controllability checker
Projection limit 3000

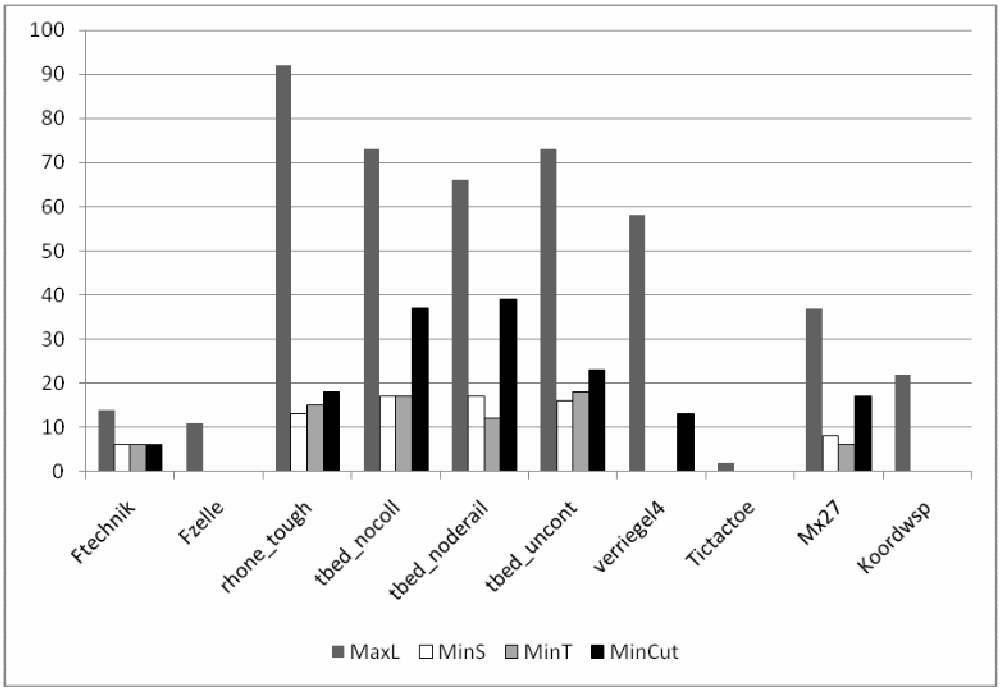


Figure 5.12: Number of projection overflows with Projection limit 3000

According to Table A1, heuristic minCut is the only heuristic that can solve all of the models. The other three heuristics cannot solve the model *tbed_ctct*. Figure 5.10 and Figure 5.12 show that heuristic maxL has a small number of the states explored by model checker but the largest number of the projection overflows. Especially for small models, it has the smallest number of states. Whereas, for the model *tbed_nocoll* and *tbed_noderail*, the number of the states with maxL is about two times larger than the number of the states with minCut. As for the heuristic minS, it has faster processing time of model checking than the other three heuristics. It can be seen from figure 5.11. However, for the model *rhone_tough*, the processing time of minS is slower than the processing time of the other three heuristics. As for the heuristic minT, it has similar performance with the heuristic minS except that minT has a larger number of the states and slower processing

time of the model checking. Nevertheless, for the model *tbed_noderail*, the number of with heuristic minT is smaller than the number of states with minS. As for the heuristic minCut, its processing time of model checking is fast and close to the processing time of model checking with the heuristic minS. For the model *tbed_nocoll*, t*bed_noderail* and *mx27*, the number of states with the heuristic minCut is smaller than the number of states with the other three heuristics.

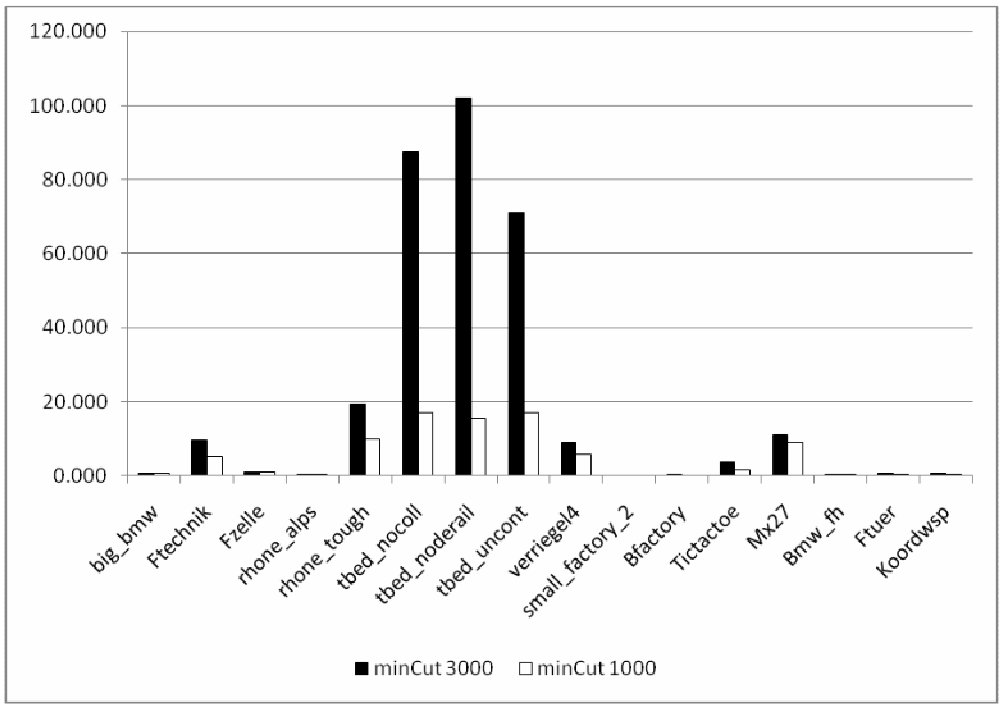


Figure 5.13: MinCut Processing Time with projection limit 3000 and 1000

For Modular controllability checker

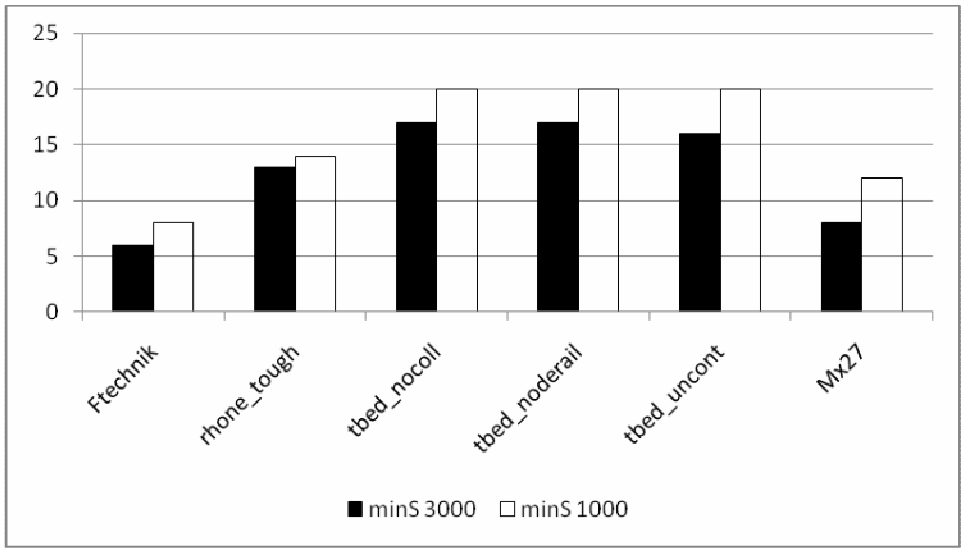


Figure 5.14: MinS number of projection overflows with
projection limit 3000 and 1000

According to Table A2, these four heuristics have similar behavior with their behavior in Table A1. With the comparing of these two tables, it shows that the processing time with larger projection limit is slower and the number of the projection overflows with larger projection limit is smaller. The reason is that large projection limit allows more projection and more projection means more processing time. At the same time, the large projection limit allows more candidates pass the projection without selecting the next candidate, so its number of projection overflows is small. It can be seen from Figure 5.13 and Figure 5.14. Composing and projection are expensive and slow. Therefore, if the compositional model checker does more composing and projection, the processing time of model checking will be longer. However, if the compositional model checker does less composing and projection, then the model may not be small enough for the non-projection controllability checker to handle. The state space problem still exists. Consequently, for small models, the projection limit should be smaller

while for large models, the projection limit should be larger.

Table A3 and Table A4 in the appendix show the results of controllability check with projection limit three thousand and one thousand respectively for the BDD controllability checker. Table A5 and Table A6 in the appendix show the results of controllability check with projection limit three thousand and one thousand respectively for the Native controllability checker. The four heuristics for these two non-projection controllability checkers have similar behaviors to the heuristics for the Modular controllability checker. However, as for Native controllability checker, all the examples can be solved by these four heuristics.

5.7.3 Language Inclusion Check Results

Table A7, A8, A9 and A10 in the appendix show the results of language inclusion check with projection limit three thousand for the BDD language inclusion checker. Table A11, A12, A13, and A14 show the results of language inclusion check with projection limit one thousand for the BDD language inclusion checker. Data is summarized in Figure 5.15, 5.16, 5.17, and 5.18.

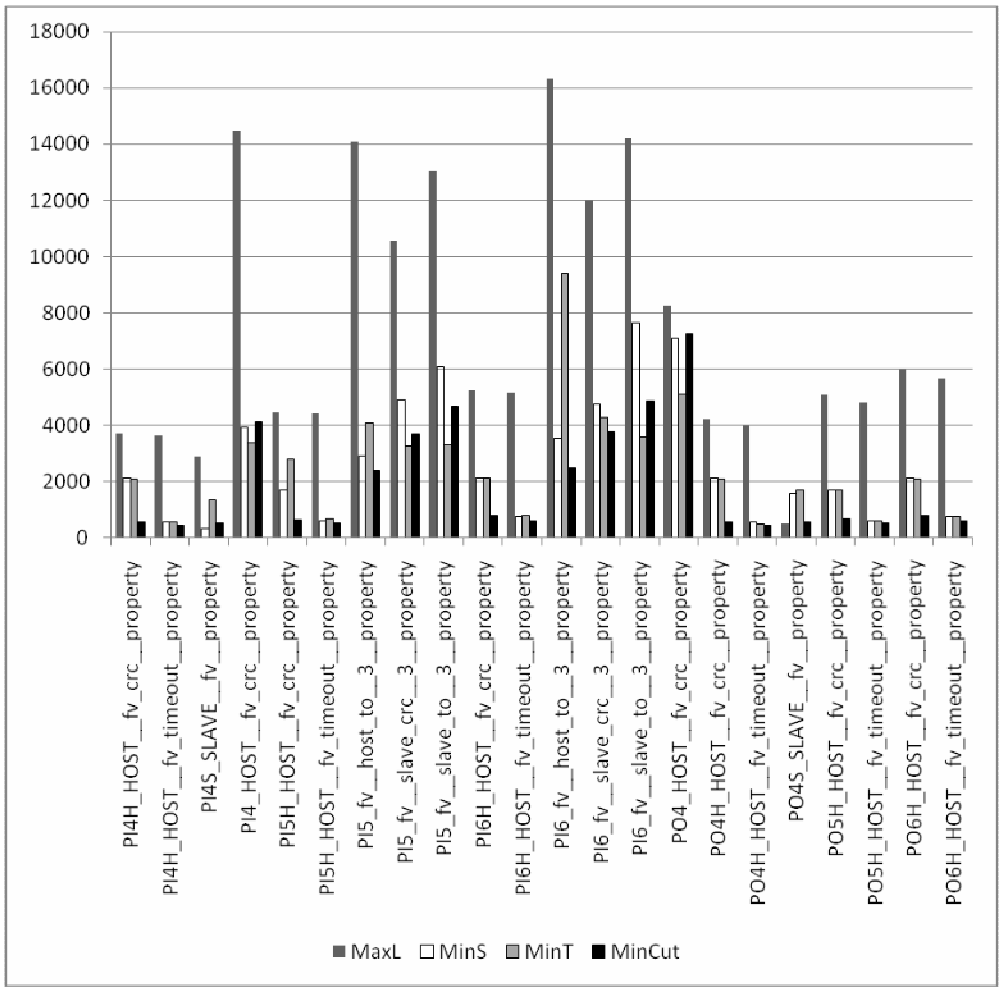


Figure 5.15: Number of states of the new automata for

BDD language inclusion check with projection limit 3000

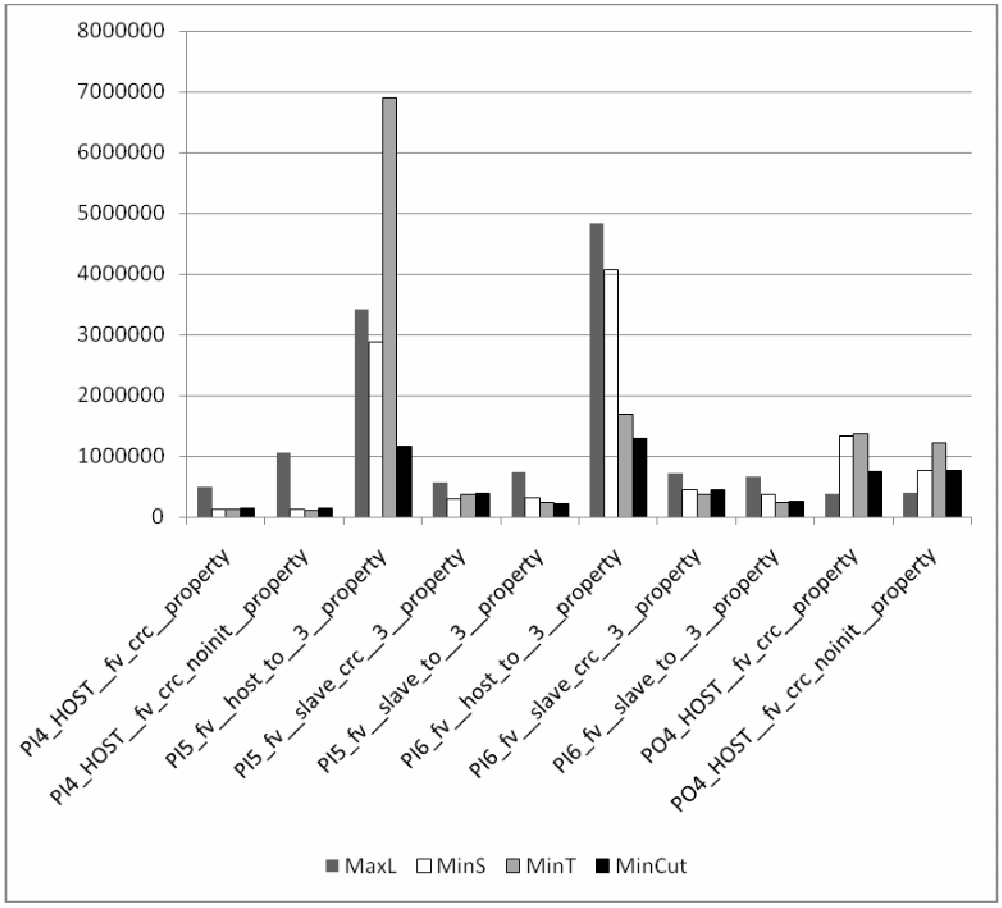


Figure 5.16: Peak Number of nodes for BDD language inclusion check with projection limit 3000

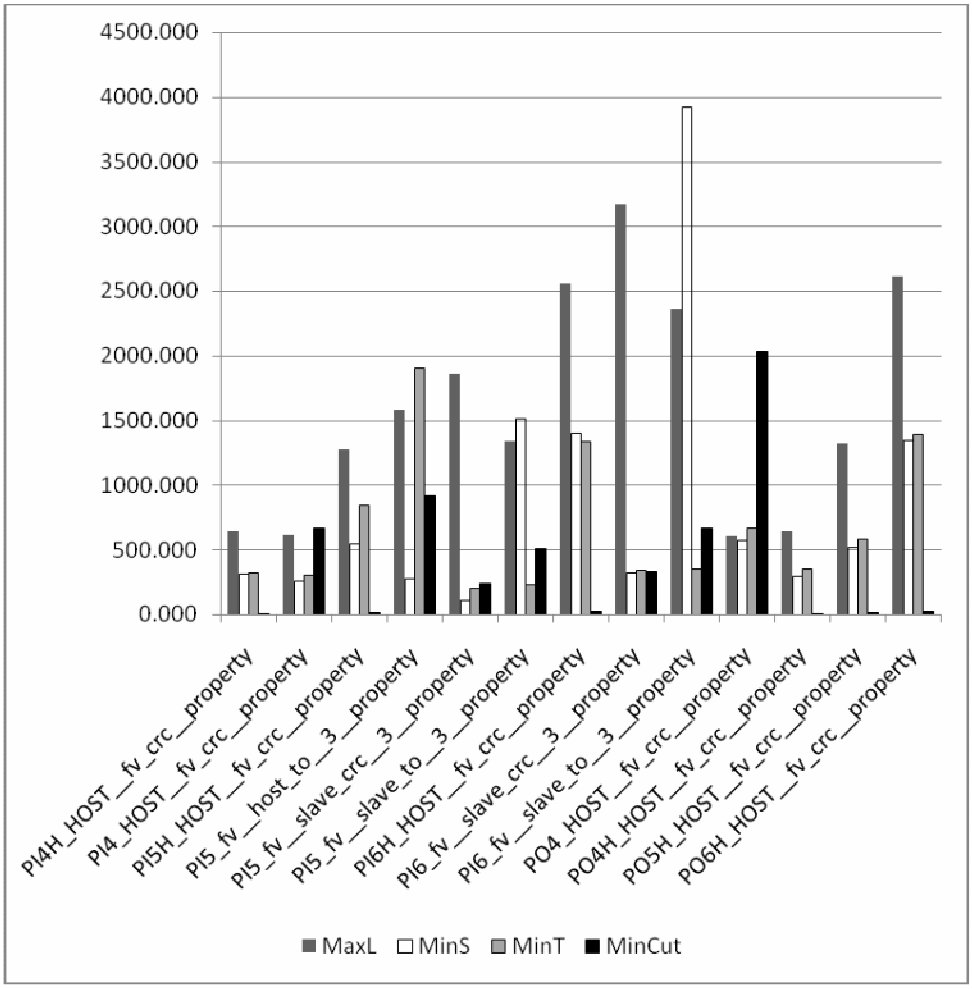


Figure 5.17: Processing time for BDD language inclusion check with projection limit 3000

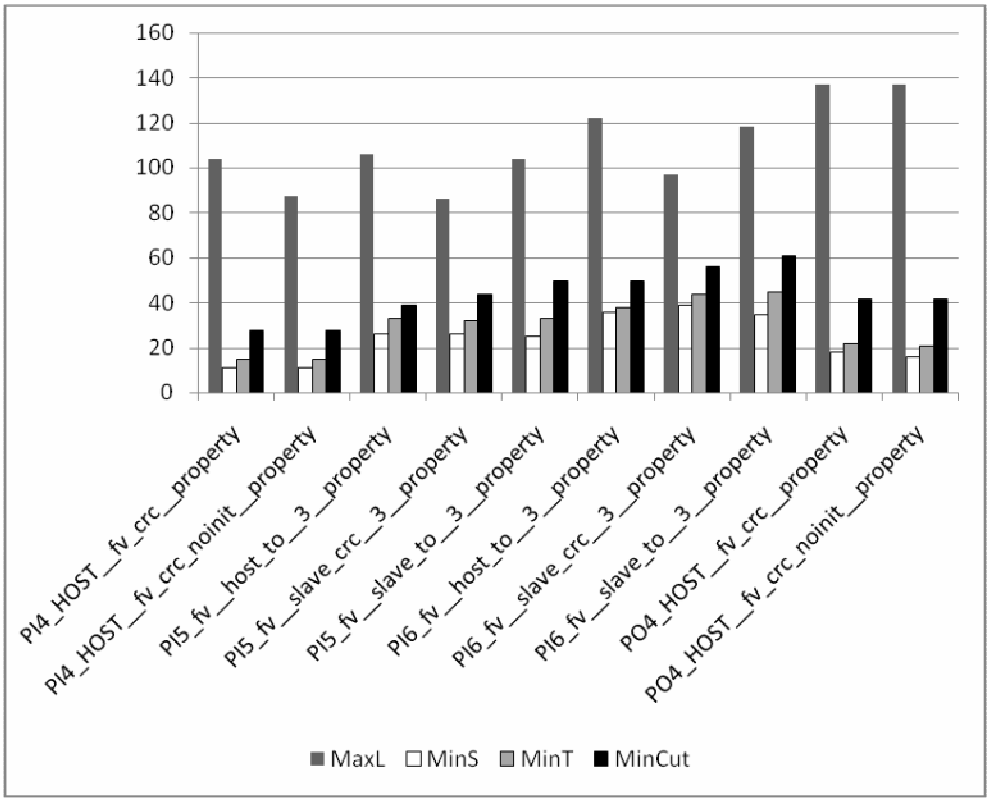


Figure 5.18: Number of projection overflows for BDD language inclusion check with projection limit 3000

According to Table A7, A8, A9 and A10, heuristic maxL is the heuristic that can solve most of the models. It cannot solve the model *profisafe_i4* with the property *SLAVE__fv__property*. Figure 5.15, 5.16, 5.17 and 5.18 show that the heuristic maxL has the largest number of the states of the new automata, the largest peak number of the nodes explored by the BDD model checker, the longest processing time of model checking and the largest number of projection overflows. That is because the compositional model checker can project more events out and do more composing and projection with maxL. Whereas, for the model *profisafe_o4* with the properties *HOST__fv_crc__property* and *HOST__fv_crc_noinit__property*, the peak number of the nodes with maxL is

about half less than the peak number of the nodes with the other three heuristics. As for the heuristic minS, it has faster processing time of model checking than the other three heuristics for most of the models according to Figure 5.17. Especially for the large examples, it has faster processing time. For example, the processing time of model checking with minS for the model *profisafe_i5* with the properties *fv__host_to__3__property* and *fv__host_to__3r__property*, is the fastest. However, for some smaller models as *profisafe_i4_host* with the properties *HOST__fv_crc__property* and *HOST__fv_crc_noinit__property*, the processing time with minS is slower than the processing time with minCut. According to Figure 5.18, the number of projection overflows with minS is the smallest. That is because that minS selects the small candidates first, it is not easy to exceed the projection limit. Unfortunately, heuristic minS is the heuristic which can solve the least models according to Table A7, A8, A9 and A10. As for the heuristic minT, it has similar performance to heuristic minS except that minT has slower processing time of model checking. Nevertheless, for the model *profisafe_i5_host* with the property *HOST__fv_crc_noinit__property*, the processing time of model checking with heuristic minT is faster than the processing time of model checking with minS. As for the heuristic minCut, its processing time of model checking is faster than the processing time of model checking with minS for small models and slower than the processing time of model checking with minS for large models. For the model *profisafe_i4* with the property *SLAVE__fv__property*, it cannot complete the composing part.

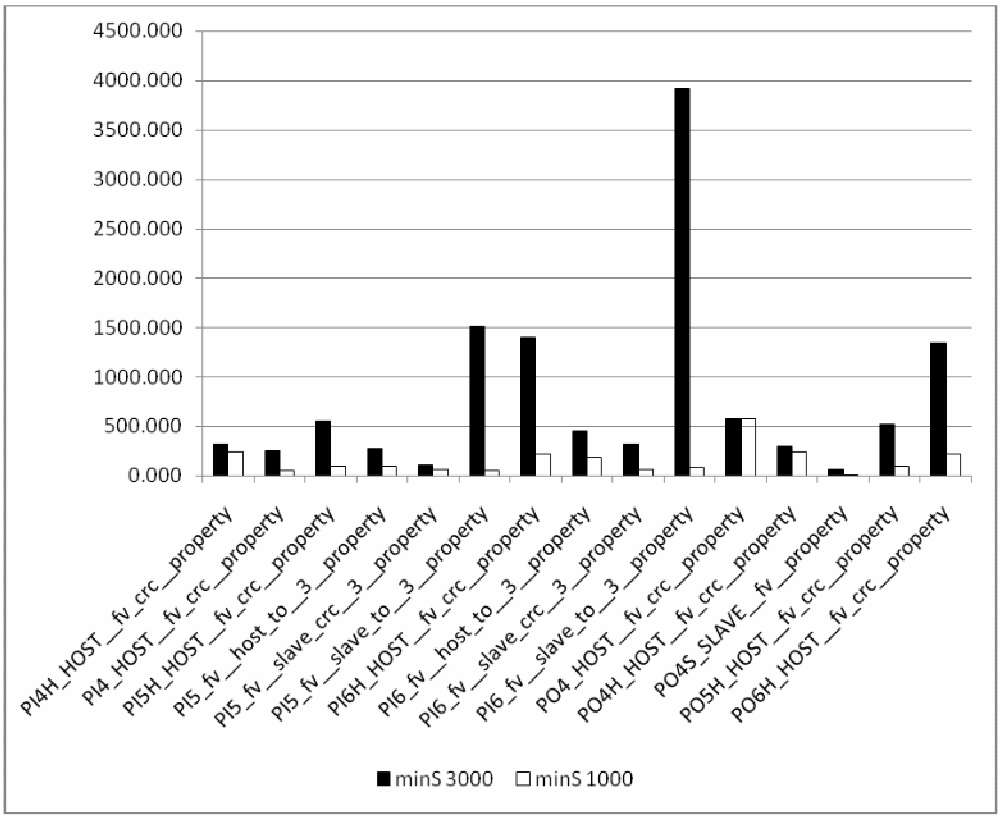


Figure 5.19: MinS Processing time with projection limit 3000 and 1000 for BDD language inclusion checker

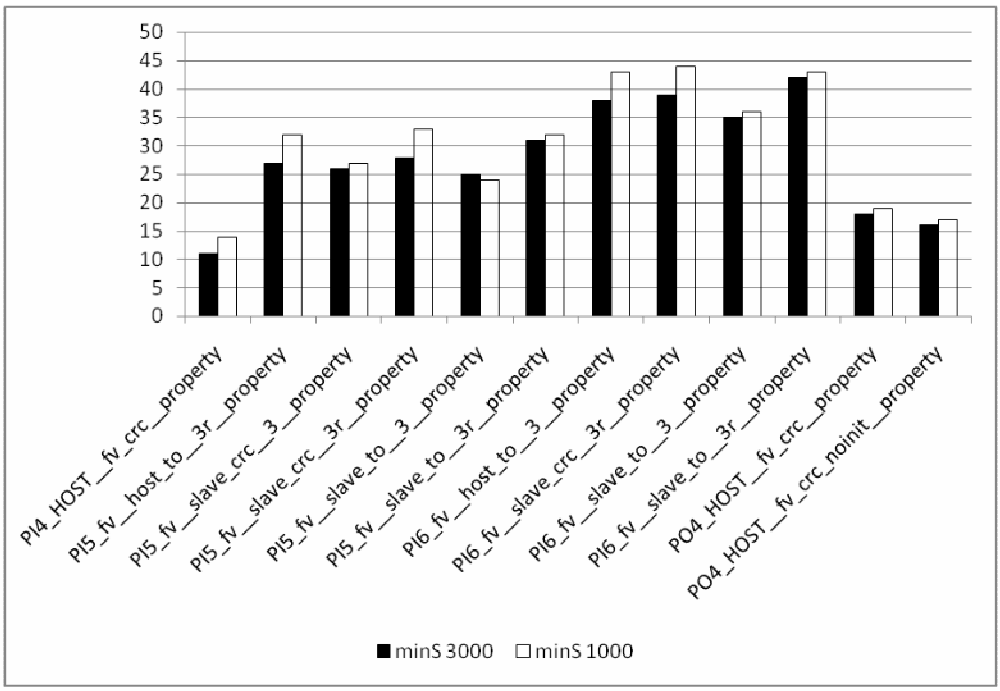


Figure 5.20: MinS number of projection overflows with projection limit 3000 and 1000

According to Table A11, A12, A13, and A14, these four heuristics have similar behavior with their behaviors in Table A7, A8, A9 and A10. With the comparing of these tables, it shows that the processing time of model checking with larger projection limit is slower than the processing time of model checking with smaller projection limit according to Figure 5.19. That is because that composing and projection are expensive and slow and large projection limit allows more projection. Also, the number of projection overflows with small projection limit is large for most of the examples according to Figure 5.20. Since the non-projection BDD model checker can handle the model with around ten million nodes well, choosing the proper projection limit and making the number of nodes of the model after composing and projection to be around ten million is still a challenge.

Table A15, A16, A17 and A18 show the results of language inclusion check with projection limit three thousand for the Modular language inclusion checker. Table A19, A20, A21, and A22 show the results of language inclusion check with projection limit one thousand for the Modular language inclusion checker. The four heuristics for the non-projection Modular language inclusion checker have similar behaviors to the heuristics for the BDD language inclusion checker except that the Modular model checker solves fewer examples.

Chapter 6

Conclusion

This project presents an enhanced compositional verification method to improve the performance of the verification of safety properties of models. This method tries to reduce the complexity and size of the model by composing a proper subset of the automata of the model and projecting some events out iteratively, and then obtain a smaller and less complex model. Experimental results show that the method can reduce the complexity and size of models indeed and improve the performance of verification of safety properties. Especially for the large industrial examples, this method can cope quite well with them. The example *profisafe_i6*, which was never verified for the language inclusion check before, is verified by this method.

The enhanced compositional verification method employs a two-step method for choosing a set of automata to compose. There are four heuristics, which are heuristic maxL, minS minT and minCut, at the second step of automata selection. According to the experimental data, these heuristics are good at checking different examples. Heuristic maxL can verify most of the large models for language inclusion check, while heuristic minCut is the only heuristic which can verify all of the models considered in the test suite for controllability check with all different model checkers. Heuristic minS and minT have similar behaviors and are stable at checking large models in short processing time for both controllability check and language inclusion check.

There are still some ways that can improve the compositional verification of safety properties method. How to select the maximum state limit for projection is the first challenge. A small maximum state limit for projection can speed up the checking, but the model may not be simplified enough for checking. A large maximum state limit for projection can simplify the model more, but it will cost more time. As for heuristic minS and minT, the prediction of the number of states and the number of transitions of the synchronous product of a set of automata can be more accurate in order to improve the automata selection by taking some more reliable new algorithms. Last but not least, different heuristics for the automata selection and new effective automaton simplification approaches can also be studied. For example, automatically select the proper heuristic for different models, or find a new simple way to represent models.

References

[AFF02] Knut Åkesson, Martin Fabian, Hugo Flordal, "*Exploiting modularity for synthesis and verification of supervisors*", *Proc. 15th IFAC World Congress on Automatic Control*, Barcelona, Spain, 2002.

[BBFLPP98] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, *"Systems and Software Verification",* Springer, 1998.

[BC94] Bertil Brandin, François Charbonnier, "*The supervisory control of the automated manufacturing system of the AIP*", *Proceedings of Rensselaer's 4th International Conference on Computer Integrated Manufacturing and Automation Technology*, pages 319-324, Troy, NY, USA, 1994.

[BMM04] B. A. Brandin, R. Malik, P. Malik, "*Incremental verification and synthesis of discrete-event systems guided by counter-examples*", *IEEE Trans. Contr. Syst. Technol.*, vol. 12, no. 3, pp. 387-401, May 2004.

[CMM05] Charles M. Macal, "*Verification and Validation*", *Workshop on "Threat Anticipation: Social Science Methods and Models",* The University of Chicago and Argonne National Laboratory, Chicago, IL, April 7-9, 2005.

[FM06] Hugo Flordal, Robi Malik, "*Modular Nonbloking Verification Using Conflict Equivalence*", *Proceedings of the 8th International Workshop on Discrete Event Systems*, Ann Arbor, Michigan, USA, pp. 100-106, July 10-12, 2006.

[GC85] Gary Chartrand, "*Introductory Graph Theory*", New York, Dover Pubns, 1985.

[HMU01] J. E. Hopcroft, R. Motwani, J. D. Ullman, "*Introduction to Automata Theory, Languages, and Computation*", Addison-Wesley, 2001.

[JS06] Jinjian Shi, "*FSM Controllability Checker: A Plug-in for Waters Toolkits*", Department of Computer Science, The University of Waikato, 2006.

[KG95] R. Kumar, V. K. Garg, "*Modeling and control of logical discrete event systems"*, Kluwer Academic Publishers, 1995.

[LL95] C. Lewerentz, T, Linder, "*Case Study 'Production Cell'*", volume 891 of *LNCS*. Springer-Verlag, 1995.

[LM96] Annette Lötzbeyer, Reinhard Mühlfeld, "*Task description of a flexible production cell with real time properties*", Technical report, FZI, Karlsruhe, Germany, 1996.

[MFMC] "*Max-flow min-cut theorem*", [Online], Available: http://en.wikipedia.org/wiki/Max-flow_min-cut_theorem

[MJD99] Musa J.D. "*Software reliability engineering"*, *McGraw* Hill, 1999.

[MM02] Robi Malik, Reinhard Mühlfeld, "*Testing the PROFIsafe protocol using automatically generated test cases based on a formally verified model*", Technical report, Siemens AG, Corporate Technology, Software and Engineering 1, Munich, Germany, 2002.

[MM03] Robi Malik, Reinhard Mühlfeld, "*A case study in verification of UML statecharts: the PROFIsafe protocol*", *Journal of Universal Computer Science*, 9(2):138-151, February, 2003.

[MS05] Michael Sipser, *"Introduction to the Theory of Computation"*, Course Technology, 2005.

[OB97] O. Balci. "*Verification validation and accreditation of simulation models*", In D. H. Withers, B. L. Nelson, S. Andradóttir, and K. J. Healy, editors, *Proc. of the 29th WSC*, pages 135– 141, 1997.

[PD00] Petra Dietrich, "*Projekt BMW E65 CAS – FH-Master – eine Modellierung in DCD*", Technical report, Siemens AG, Corporate Technology, Software and Engineering 4, Munich, Germany, 2000.

[PM03] Petra Malik, "*From supervisory Control to Nonblocking Controllers for Discrete Event Systems*", PhD thesis, University of Kaiserslautern, Kaiserslautern, Germany, 2003.

[PN02] Profibus Nutzerorganisation e. V, "*PROFIsafe – profile for safety technology, version 1.12*", 2002.

[PS08] Peter Stringer, "*Using Machine Learning Techniques to Predict Automata Sizes*", Department of Computer Science, The University of Waikato, 2008.

[REB86] Randal E. Bryant, "*Graph-Based Algorithms for Boolean Function Manipulation*", *IEEE Transactions on Computers*, Vol. C - 35, No. 8, pp 677–691, 1986.

[RJL96] R. J. Leduc, "*PLC implementation of a DES supervisor for a manufacturing testbed: An implementation perspective*", Master's Thesis, Department of Electrical Engineering, University of Toronto, Ontario, Canada, 1996.

[RW89] Peter J. G. Ramadge , W. Murray Wonham, *"The control of discrete event systems", Proc. of IEEE*, vol. 77, no. 1, pp. January 81–98, 1989.

[SW07] Simon Ware, "*Modular Finite-State Machine Analysis*", Department of Computer Science, The University of Waikato, 2007.

[WM08] Simon Ware, Robi Malik, "*The Use of Language Projection for Compositional Verification of Discrete Event System*", *Proc. 9th International Workshop on Discrete Event Systems*, WODES'08, 322-327, Göteborg, Sweden, 28-30 May 2008.

[WS] "WATERS", [Online], Available: http://www.cs.waikato.ac.nz/~robi/waters/

Appendix

Compositional Verification Results

Model	MaxL			MinS			MinT			MinCut		
	States	Time(s)	O	States	Time(s)	O	States	Time(s)	O	States	Time(s)	O
big_bmw	1	1.517	0	6	0.629	0	6	1.420	0	7	0.688	0
Ftechnik	8	16.211	14	18	9.900	6	23	12.845	6	18	9.815	6
Fzelle	2	34.302	11	4	1.821	0	4	7.605	0	5	1.191	0
rhone_alps	2	1.165	0	2	0.947	0	2	0.923	0	2	0.659	0
rhone_tough	18181	59.214	92	496876	130.738	13	117128	25.506	15	56560	19.691	18
tbed_ctct		O			O			O		4713148	51.425	35
tbed_nocoll	7553	93.034	73	7227	31.695	17	7385	95.818	17	4994	87.560	37
tbed_noderail	7023	82.716	66	7211	31.632	17	5521	90.784	12	3814	102.287	39
tbed_uncont	2640	95.476	73	2612	37.282	16	2612	79.759	18	2664	71.136	23
verriegel4	1	124.062	58	2	4.321	0	2	5.144	0	2	9.245	13
small_factory_2	2	0.238	0	2	0.238	0	2	0.237	0	2	0.240	0
Bfactory	1	0.496	0	1	0.250	0	1	0.271	0	1	0.392	0
Tictactoe	1	68.532	2	1	1.385	0	1	1.215	0	1	3.718	0
Mx27	10548	115.273	37	15465	6.206	8	17891	10.937	6	3752	11.117	17
Bmw_fh	1	1.399	0	1	0.483	0	1	0.812	0	2	0.623	0
Ftuer	1	0.715	0	1	0.344	0	1	0.378	0	1	0.693	0
Koordwsp	1	29.107	22	2	0.956	0	2	1.940	0	2	0.686	0

Table A1: Modular Controllability Check with Projection limit 3000

Model	MaxL			MinS			MinT			MinCut		
	States	Time(s)	O	States	Time(s)	O	States	Time(s)	O	States	Time(s)	O
big_bmw	1	1.362	0	6	0.627	0	6	1.127	0	7	0.688	0
Ftechnik	8	20.909	27	24	5.236	8	26	7.366	7	24	5.393	8
Fzelle	2	7.427	15	4	1.579	0	5	3.571	0	5	1.194	0
rhone_alps	2	1.058	0	2	0.570	0	2	1.008	0	2	0.652	0
rhone_tough	132240	18.324	75	366141	8.088	14	122965	12.460	18	326990	10.028	27
tbed_ctct		O			O			O		2849052	26.337	43
tbed_nocoll	21121	28.998	88	43976	15.190	20	44348	60.019	22	33981	17.128	27
tbed_noderail	19942	28.603	84	43572	14.831	20	16649	49.395	25	31328	15.824	22
tbed_uncont	5082	30.266	88	8083	15.712	20	8083	58.803	21	9419	17.310	27
verriegel4	2547	25.131	62	2	3.388	0	2	5.065	1	2	5.767	14
small_factory_2	2	0.255	0	2	0.243	0	2	0.242	0	2	0.234	0
Bfactory	1	0.246	0	1	0.252	0	1	0.255	0	1	0.263	0
Tictactoe	6	7.766	9	1	0.966	0	1	1.537	0	1	1.662	6
Mx27	15654	11.157	34	14835	2.545	12	5661	3.942	8	3752	9.135	17
Bmw_fh	1	1.220	0	1	0.670	0	1	0.589	0	2	0.607	0
Ftuer	1	0.497	0	1	0.382	0	1	0.793	0	1	0.397	0
Koordwsp	1	16.283	29	2	0.968	0	2	1.655	1	2	0.667	0

Table A2: Modular Controllability Check with Projection limit 1000

Model	MaxL			MinS			MinT			MinCut		
	Nodes	Time(s)	O	Nodes	Time(s)	O	Nodes	Time(s)	O	Nodes	Time(s)	O
big_bmw	1	1.509	0	1	0.657	0	1	1.153	0	1	0.711	0
Ftechnik	1	16.822	14	76237	9.142	6	78481	10.337	6	76237	9.374	6
Fzelle	1	35.960	11	1	1.606	0	1	7.771	0	1	1.246	0
rhone_alps	1	1.133	0	1	0.611	0	1	1.044	0	1	0.759	0
rhone_tough	113893	62.334	92	515529	182.698	13	261834	30.160	15	191721	27.026	18
tbed_ctct		O			O			O		7970262	1500.452	35
tbed_nocoll	422357	103.722	73	397028	38.297	17	402697	102.776	17	207488	78.792	37
tbed_noderail	385741	93.119	66	394511	34.603	17	155854	100.902	12	114535	90.646	39
tbed_uncont	330219	96.228	73	320476	31.015	16	435543	77.576	18	381368	49.047	23
verriegel4	1	111.915	58	1	3.648	0	1	5.181	0	1	8.562	13
small_factory_2	1	0.268	0	1	0.275	0	1	0.272	0	1	0.273	0
Bfactory	1	0.279	0	1	0.291	0	1	0.324	0	1	0.328	0
Tictactoe	61	66.709	2	61	0.971	0	58	1.173	0	61	3.040	0
Mx27	254699	126.005	37	93156	7.010	8	273033	16.688	6	51856	11.613	17
Bmw_fh	1	1.318	0	1	0.586	0	1	0.603	0	1	0.686	0
Ftuer	1	0.521	0	1	0.388	0	1	0.392	0	1	0.377	0
Koordwsp	1	28.872	22	1	0.993	0	1	1.638	0	1	0.846	0

Table A3: BDD Controllability Check with Projection limit 3000

Model	MaxL			MinS			MinT			MinCut		
	Nodes	Time(s)	O	Nodes	Time(s)	O	Nodes	Time(s)	O	Nodes	Time(s)	O
big_bmw	1	1.864	0	1	0.648	0	6	1.624	0	1	0.702	0
Ftechnik	1	23.031	27	40952	4.455	8	87296	7.061	7	40952	5.303	8
Fzelle	1	8.937	15	1	1.609	0	1	3.578	0	1	1.276	0
rhone_alps	1	1.091	0	1	0.606	0	1	1.021	0	1	0.673	0
rhone_tough	260982	28.161	75	133363	10.434	14	142188	14.858	24	195817	14.756	27
tbed_ctct		O			O			O		8354751	1685.074	43
tbed_nocoll	296563	34.667	88	326664	25.436	20	337156	69.783	21	309319	23.942	27
tbed_noderail	305325	39.894	84	324322	21.749	20	277956	68.706	25	238891	20.367	22
tbed_uncont	194459	30.444	88	235448	17.954	20	227293	58.803	21	208754	17.836	27
verriegel4	29053	25.375	62	1	4.313	3	1	10.055	2	1	6.297	14
small_factory_2	1	0.249	0	1	0.276	0	1	0.266	0	1	0.246	0
Bfactory	1	0.267	0	1	0.520	0	1	0.268	0	1	0.261	0
Tictactoe	11652	6.922	9	59	0.882	0	58	1.141	0	61	1.962	6
Mx27	103812	11.845	34	17551	2.689	12	52808	4.642	8	55581	9.688	17
Bmw_fh	1	1.235	0	1	0.556	0	1	0.961	0	1	0.595	0
Ftuer	1	0.515	0	1	0.415	0	1	0.386	0	1	0.548	0
Koordwsp	1	17.056	29	1	0.989	0	1	1.345	1	1	0.681	0

Table A4: BDD Controllability Check with Projection limit 1000

Model	MaxL			MinS			MinT			MinCut		
	States	Time(s)	O	States	Time(s)	O	States	Time(s)	O	States	Time(s)	O
big_bmw	0	1.439	0	0	0.612	0	0	1.124	0	0	0.663	0
Ftechnik	1	14.686	14	1	8.100	6	2	9.480	6	1	8.040	6
Fzelle	0	30.008	11	0	1.688	0	0	3.746	0	0	1.153	0
rhone_alps	0	1.112	0	0	0.869	0	0	0.911	0	0	0.638	0
rhone_tough	16680	51.848	92	289035	121.359	13	153587	107.832	14	56538	18.032	18
tbed_ctct	1034643	53.217	77	612720	20.176	30	609022	78.987	32	587347	22.030	35
tbed_nocoll	7095	69.884	73	6780	20.891	17	5573	95.946	15	4736	71.079	37
tbed_noderail	6616	64.888	66	6764	20.083	17	5161	85.930	12	3624	86.691	39
tbed_uncont	2182	73.838	73	2210	22.508	16	2210	74.068	19	2169	39.082	23
verriegel4	0	109.927	58	0	3.576	0	0	4.910	0	0	8.601	13
small_factory_2	0	0.217	0	0	0.218	0	1	0.218	0	0	0.221	0
Bfactory	1	0.452	0	1	0.251	0	1	0.250	0	1	0.302	0
Tictactoe	1	65.002	2	1	0.901	0	58	1.173	0	1	3.135	0
Mx27	395	112.541	37	541	5.337	8	529	8.652	6	333	11.069	17
Bmw_fh	0	1.249	0	0	0.594	0	0	0.830	0	1	0.671	0
Ftuer	0	0.527	0	0	0.334	0	0	0.387	0	0	0.377	0
Koordwsp	0	30.439	22	0	0.934	0	0	1.626	0	0	0.846	0

Table A5: Native Controllability Check with Projection limit 3000

Model	MaxL			MinS			MinT			MinCut		
	States	Time(s)	O	States	Time(s)	O	States	Time(s)	O	States	Time(s)	O
big_bmw	0	1.348	0	0	0.605	0	0	1.163	0	0	0.660	0
Ftechnik	1	21.612	27	2	3.784	8	2	5.379	7	2	3.878	8
Fzelle	0	7.400	15	0	1.555	0	0	3.665	0	0	1.180	0
rhone_alps	0	1.040	0	0	0.552	0	0	0.990	0	0	0.635	0
rhone_tough	63979	17.356	75	211479	6.491	14	122949	12.384	27	194327	7.723	27
tbed_ctct	642589	22.320	109	646329	10.281	34	651172	65.901	40	626335	12.982	43
tbed_nocoll	7933	23.998	88	7263	10.931	20	7417	61.152	20	7513	13.486	27
tbed_noderail	7528	23.619	84	7247	9.894	20	7247	51.970	20	7012	11.757	22
tbed_uncont	2445	25.020	88	2418	10.613	20	2477	53.889	23	2519	13.260	27
verriegel4	2536	24.448	62	0	3.408	3	0	4.985	1	0	5.715	14
small_factory_2	0	0.229	0	0	0.227	0	0	0.239	0	0	0.223	0
Bfactory	1	0.433	0	1	0.402	0	1	0.240	0	1	0.287	0
Tictactoe	4	6.425	9	1	1.120	0	1	1.407	0	1	2.011	6
Mx27	495	10.753	34	487	2.382	12	439	3.660	8	333	9.222	17
Bmw_fh	0	1.478	0	0	0.530	0	0	0.755	0	1	0.569	0
Ftuer	0	0.859	0	0	0.335	0	0	0.382	0	0	0.411	0
Koordwsp	0	17.865	29	0	0.992	0	0	1.280	1	0	0.736	0

Table A6: Native Controllability Check with Projection limit 1000

Model	Property		MaxL				MinS			
			States	Nodes	Time(s)	O	States	Nodes	Time(s)	O
profisafe_i4_host	HOST__fv_crc__property	F	3703	1	638.371	6	2108	1	320.586	0
	HOST__fv_crc_noinit__property	T	3865	1	578.310	6	2896	1	550.825	0
	HOST__fv_timeout__property	T	3644	1	621.217	6	541	1	7.333	0
profisafe_i4_slave	SLAVE__fv__property	T	2906	1	1182.728	3	314	1	1.431	0
profisafe_i4	HOST__fv_crc__property	F	14487	483672	615.524	104	3933	127134	260.390	11
	HOST__fv_crc_noinit__property	F	9258	1056551	635.400	87	4147	125906	313.274	11
	HOST__fv_timeout__property	T	9957	1205408	1015.782	92	2935	30239	162.264	9
	SLAVE__fv__property	T			O		3660	31655	134.097	9
profisafe_i5_host	HOST__fv_crc__property	F	4470	1	1277.852	6	1680	34282	551.681	1
	HOST__fv_crc_noinit__property	T	4675	1	1298.521	6	2237	35673	789.305	1
	HOST__fv_timeout__property	T	4403	1	1366.733	6	592	1	9.037	0
profisafe_i5	fv__host_crc__3__property	F	16188	2293793	1683.469	82			O	
	fv__host_crc__3r__property	F	10567	8044925	2021.481	80			O	
	fv__host_to__3__property	F	14078	3420643	1579.686	106	2921	2883929	277.003	26
	fv__host_to__3r__property	F	11378	2207089	1308.215	98	4372	3638864	880.442	27
	fv__slave_crc__3__property	F	10577	569829	1854.939	86	4914	309917	114.149	26
	fv__slave_crc__3r__property	F	10263	453107	1366.198	91	5297	494126	240.795	28
	fv__slave_to__3__property	F	13019	732655	1338.195	104	6096	313276	1513.757	25
	fv__slave_to__3r__property	F	11232	912213	1326.380	108	4245	786683	671.978	31
profisafe_i6_host	HOST__fv_crc__property	F	5249	1	2555.211	6	2128	42091	1407.997	1
	HOST__fv_crc_noinit__property	T	5669	1	2529.207	7	1872	1	313.336	2
	HOST__fv_timeout__property	T	5174	1	2558.138	6	729	1	13.695	0
profisafe_i6	fv__host_crc__3__property	F	16147	9785240	2976.696	94			O	
	fv__host_crc__3r__property	F	9088	5019068	2206.985	91			O	
	fv__host_to__3__property	F	16321	4834128	2964.692	122	3532	4077851	1101.245	36
	fv__host_to__3r__property	F	10046	1469791	2295.577	109	4711	2871724	462.235	38
	fv__slave_crc__3__property	F	12000	717377	3170.970	97	4744	453502	326.417	39
	fv__slave_crc__3r__property	F	11686	571632	2463.972	102	4710	537557	298.882	39
	fv__slave_to__3__property	F	14203	654513	2358.595	118	7646	379971	3919.607	35
	fv__slave_to__3r__property	F	12166	745190	2396.362	121	3658	826984	740.206	42

Table A7: BDD Language Inclusion Check with Heuristic maxL and minS
for PROFIsafe Input-Slave Models
Projection limit 3000

Model	Property		MinT				MinCut			
			States	Nodes	Time(s)	O	States	Nodes	Time(s)	O
profisafe_i4_host	HOST__fv_crc__property	F	2098	1	324.447	0	550	1	12.851	0
	HOST__fv_crc_noinit__property	T	2254	1	347.531	0	951	1	35.226	0
	HOST__fv_timeout__property	T	542	1	11.075	0	441	1	6.907	0
profisafe_i4_slave	SLAVE__fv__property	T	1342	1	5.771	0	519	1	4.302	0
profisafe_i4	HOST__fv_crc__property	F	3383	129274	306.010	15	4119	145937	669.678	28
	HOST__fv_crc_noinit__property	F	3999	116717	316.978	15	4269	141994	663.279	28
	HOST__fv_timeout__property	T	3279	29165	221.536	19			O	
	SLAVE__fv__property	T	3755	31570	141.162	15			O(C)	
profisafe_i5_host	HOST__fv_crc__property	F	2772	1	846.816	0	646	1	20.991	0
	HOST__fv_crc_noinit__property	T	1031	43026	190.276	1	1077	1	59.577	0
	HOST__fv_timeout__property	T	659	1	18.220	0	495	1	9.374	0
profisafe_i5	fv__host_crc__3__property	F	3004	1295599	328.279	23	1646	5898284	16667.012	43
	fv__host_crc__3r__property	F			O				O	
	fv__host_to__3__property	F	4069	6906086	1907.793	33	2381	1166786	919.688	39
	fv__host_to__3r__property	F	7246	3631893	3720.813	28	5708	5468201	13309.945	40
	fv__slave_crc__3__property	F	3261	384974	201.219	32	3702	392745	245.634	44
	fv__slave_crc__3r__property	F	4024	349096	222.795	26	3668	494369	229.597	44
	fv__slave_to__3__property	F	3295	251424	233.142	33	4660	220396	506.892	50
	fv__slave_to__3r__property	F	4510	332135	299.546	28	5316	175119	207.378	48
profisafe_i6_host	HOST__fv_crc__property	F	2124	42091	1343.483	1	780	1	32.148	0
	HOST__fv_crc_noinit__property	T	1234	47455	401.795	1	1245	1	95.200	0
	HOST__fv_timeout__property	T	768	1	22.366	0	583	1	13.382	0
profisafe_i6	fv__host_crc__3__property	F			O		2778	6342190	3479.545	53
	fv__host_crc__3r__property	F			O		2212	7460553	1549.695	57
	fv__host_to__3__property	F	9424	1696732	8851.137	38	2477	1297223	1101.494	50
	fv__host_to__3r__property	F	4918	1813103	564.495	36	6158	5806199	2875.322	54
	fv__slave_crc__3__property	F	4280	380113	340.435	44	3777	458345	330.785	56
	fv__slave_crc__3r__property	F	4064	513234	321.591	43	3743	568779	334.990	56
	fv__slave_to__3__property	F	3587	238732	353.987	45	4868	267722	670.341	61
	fv__slave_to__3r__property	F	4011	193434	341.865	36	5603	219804	348.097	60

Table A8: BDD Language Inclusion Check with Heuristic minT and minCut
for PROFIsafe Input-Slave Models
Projection limit 3000

Model	Property		MaxL				MinS			
			States	Nodes	Time(s)	O	States	Nodes	Time(s)	O
profisafe_o4	HOST__fv_crc__property	F	8259	371531	602.369	137	7079	1336962	575.188	18
	HOST__fv_crc_noinit__property	F	8229	387799	602.785	137	8767	769108	2037.814	16
	HOST__fv_timeout__property	O								
	SLAVE__fv__property	O								
profisafe_o4_host	HOST__fv_crc__property	F	4216	1	642.252	6	2112	1	295.411	0
	HOST__fv_crc_noinit__property	T	4209	1	653.892	6	2900	1	497.484	0
	HOST__fv_timeout__property	T	3995	1	652.045	6	545	1	7.908	0
profisafe_o4_slave	SLAVE__fv__property	T	501	1	20.465	3	1569	1	62.745	0
profisafe_o5	fv__host_crc__3__property	O								
	fv__host_crc__3r__property	O								
	fv__host_to__3__property	O								
	fv__host_to__3r__property	O								
	fv__slave_crc__3__property	O								
	fv__slave_crc__3r__property	O								
	fv__slave_to__3__property	O								
	fv__slave_to__3r__property	O								
profisafe_o5_host	HOST__fv_crc__property	F	5096	1	1323.265	6	1682	33191	523.873	1
	HOST__fv_crc_noinit__property	T	5088	1	1321.574	6	2239	37511	794.359	1
	HOST__fv_timeout__property	T	4826	1	1394.646	6	596	1	8.845	0
profisafe_o6	fv__host_crc__3__property	O								
	fv__host_crc__3r__property	O								
	fv__host_to__3__property	O								
	fv__host_to__3r__property	O								
	fv__slave_crc__3__property	O								
	fv__slave_crc__3r__property	O								
	fv__slave_to__3__property	O								
	fv__slave_to__3r__property	O								
profisafe_o6_host	HOST__fv_crc__property	F	5988	1	2614.525	6	2130	40253	1348.888	1
	HOST__fv_crc_noinit__property	T	5979	1	2621.277	6	1877	1	313.969	2
	HOST__fv_timeout__property	T	5669	1	2590.517	6	733	1	13.436	0

Table A9: BDD Language Inclusion Check with Heuristic maxL and minS
for PROFIsafe Output-Slave Models
Projection limit 3000

Model	Property		MinT				MinCut			
			States	Nodes	Time(s)	O	States	Nodes	Time(s)	O
profisafe_o4	HOST__fv_crc__property	F	5110	1380687	665.073	22	7253	763590	2034.983	42
	HOST__fv_crc_noinit__property	F	6547	1222231	657.400	21	7667	766613	2109.196	42
	HOST__fv_timeout__property	O								
	SLAVE__fv__property	O								
profisafe_o4_host	HOST__fv_crc__property	F	2096	1	349.709	0	552	1	11.175	0
	HOST__fv_crc_noinit__property	T	2204	1	348.969	0	953	1	36.866	0
	HOST__fv_timeout__property	T	481	1	10.170	0	443	1	6.793	0
profisafe_o4_slave	SLAVE__fv__property	T	1704	1	8.209	0	531	1	11.572	0
profisafe_o5	fv__host_crc__3__property	O								
	fv__host_crc__3r__property	O								
	fv__host_to__3__property	O								
	fv__host_to__3r__property	O								
	fv__slave_crc__3__property	O								
	fv__slave_crc__3r__property	O								
	fv__slave_to__3__property	O								
	fv__slave_to__3r__property	O								
profisafe_o5_host	HOST__fv_crc__property	F	1686	33191	583.214	1	648	1	20.749	0
	HOST__fv_crc_noinit__property	T	2293	37511	855.659	1	1079	1	53.614	0
	HOST__fv_timeout__property	T	597	1	15.352	0	497	1	9.397	0
profisafe_o6	fv__host_crc__3__property	O								
	fv__host_crc__3r__property	O								
	fv__host_to__3__property	O								
	fv__host_to__3r__property	O								
	fv__slave_crc__3__property	O								
	fv__slave_crc__3r__property	O								
	fv__slave_to__3__property	O								
	fv__slave_to__3r__property	O								
profisafe_o6_host	HOST__fv_crc__property	F	2078	40253	1394.933	1	782	1	32.291	0
	HOST__fv_crc_noinit__property	T	1235	47745	174.452	1	1247	1	90.528	0
	HOST__fv_timeout__property	T	732	1	23.992	1	585	1	14.425	0

Table A10: BDD Language Inclusion Check with Heuristic minT and minCut
for PROFIsafe Output-Slave Models
Projection limit 3000

Model	Property		MaxL				MinS			
			States	Nodes	Time(s)	O	States	Nodes	Time(s)	O
profisafe_i4_host	HOST__fv_crc__property	F	3703	1	564.391	6	1723	30157	238.383	1
	HOST__fv_crc_noinit__property	T	3975	1	556.897	6	1032	38026	39.432	1
	HOST__fv_timeout__property	T	3644	1	637.084	6	541	1	7.023	0
profisafe_i4_slave	SLAVE__fv__property	T	1285	1	228.254	5	314	1	1.391	0
profisafe_i4	HOST__fv_crc__property	F	14487	483672	584.276	94	3253	286713	56.612	14
	HOST__fv_crc_noinit__property	F	8915	326899	554.280	90	3763	134107	196.404	11
	HOST__fv_timeout__property	T	8874	800678	544.177	87	2935	30377	15.552	9
	SLAVE__fv__property	T	7406	1085058	910.069	73	3079	31817	16.180	10
profisafe_i5_host	HOST__fv_crc__property	F	4233	1	1064.578	7	1196	1	101.278	2
	HOST__fv_crc_noinit__property	T	4851	1	1186.065	8	1024	1	41.118	2
	HOST__fv_timeout__property	T	4089	1	1093.973	7	592	1	8.444	0
profisafe_i5	fv__host_crc__3__property	F	9541	2926221	1860.284	95			O	
	fv__host_crc__3r__property	F	6132	4671844	1163.551	93			O	
	fv__host_to__3__property	F	7678	826670	780.461	104	2339	759276	101.646	26
	fv__host_to__3r__property	F	6187	1269771	789.800	102	3558	2661660	191.457	32
	fv__slave_crc__3__property	F	7097	338533	891.062	84	3677	264502	69.448	27
	fv__slave_crc__3r__property	F	7084	380466	883.710	84	2856	454044	36.297	33
	fv__slave_to__3__property	F	6819	212760	831.373	103	2953	140615	54.864	24
	fv__slave_to__3r__property	F	6881	291755	730.764	103	3008	676841	552.654	32
profisafe_i6_host	HOST__fv_crc__property	F	5602	150099	4394.671	22	1413	1	223.191	2
	HOST__fv_crc_noinit__property	T	7975	33797	4616.216	29	1191	1	69.479	2
	HOST__fv_timeout__property	T	7647	1	4677.349	26	729	1	13.123	0
profisafe_i6	fv__host_crc__3__property	F			O		2086	4748171	442.137	37
	fv__host_crc__3r__property	F			O		2125	7057589	1428.294	37
	fv__host_to__3__property	F			O	127	2343	821502	143.377	37
	fv__host_to__3r__property	F	9808	4661953	4395.696	121	2768	2863354	186.742	43
	fv__slave_crc__3__property	F	9676	381990	2336.047	120	3296	471590	60.266	39
	fv__slave_crc__3r__property	F	9663	525630	2515.958	120	2964	503575	47.634	44
	fv__slave_to__3__property	F	9608	207926	2247.112	121	2631	211698	84.313	36
	fv__slave_to__3r__property	F	10058	310661	2180.365	131	3116	761753	677.633	43

Table A11: BDD Language Inclusion Check with Heuristic maxL and minS
for PROFIsafe Input-Slave Models
Projection limit 1000

Model	Property		MinT				MinCut			
			States	Nodes	Time(s)	O	States	Nodes	Time(s)	O
profisafe_i4_host	HOST__fv_crc__property	F	992	1	48.068	2	550	1	10.538	0
	HOST__fv_crc_noinit__property	T	882	38535	77.495	1	951	1	36.794	0
	HOST__fv_timeout__property	T	591	1	10.176	0	441	1	7.539	0
profisafe_i4_slave	SLAVE__fv__property	T	1067	1	35.729	1	591	1	3.877	1
profisafe_i4	HOST__fv_crc__property	F	3400	122990	246.566	14	3735	155666	278.397	27
	HOST__fv_crc_noinit__property	F	3015	378164	123.106	16	3885	157755	254.831	27
	HOST__fv_timeout__property	T							O	
	SLAVE__fv__property	T							O(C)	
profisafe_i5_host	HOST__fv_crc__property	F	717	22665	76.294	1	646	1	18.188	0
	HOST__fv_crc_noinit__property	T	884	1	44.227	1	1077	1	51.674	0
	HOST__fv_timeout__property	T	752	1	17.392	0	495	1	9.341	0
profisafe_i5	fv__host_crc__3__property	F							O	
	fv__host_crc__3r__property	F							O	
	fv__host_to__3__property	F	4003	751127	174.373	28	3824	683644	80.802	53
	fv__host_to__3r__property	F	2158	9961779	3075.516	36	2260	1649999	4047.174	39
	fv__slave_crc__3__property	F	3819	370815	182.814	26	3160	327922	67.954	46
	fv__slave_crc__3r__property	F	3916	592446	191.659	30	2464	501949	48.845	45
	fv__slave_to__3__property	F	3016	148900	148.702	26	2954	197023	57.203	45
	fv__slave_to__3r__property	F	3812	247633	189.716	31	3397	272057	281.280	47
profisafe_i6_host	HOST__fv_crc__property	F	1417	1	211.925	1	780	1	37.716	0
	HOST__fv_crc_noinit__property	T	1032	1	71.258	3	1245	1	87.548	0
	HOST__fv_timeout__property	T	828	1	19.579	0	583	1	12.773	0
profisafe_i6	fv__host_crc__3__property	F					2778	6357354	3484.937	53
	fv__host_crc__3r__property	F					2212	7460553	1467.243	57
	fv__host_to__3__property	F	3101	624052	223.131	42	3166	761484	88.541	66
	fv__host_to__3r__property	F	2379	1617974	1495.220	37	2356	1714203	1653.996	50
	fv__slave_crc__3__property	F	3330	434467	207.969	39	2504	540115	64.476	55
	fv__slave_crc__3r__property	F	2449	737531	229.534	35	2172	487316	48.145	56
	fv__slave_to__3__property	F	2728	268848	215.914	35	2619	426883	45.017	55
	fv__slave_to__3r__property	F	3380	381872	214.689	43	3062	501782	338.100	57

Table A12: BDD Language Inclusion Check with Heuristic minT and minCut
for PROFIsafe Input-Slave Models
Projection limit1000

Model	Property		MaxL				MinS			
			States	Nodes	Time(s)	O	States	Nodes	Time(s)	O
profisafe_o4	HOST__fv_crc__property	F	7099	503418	529.375	132	3433	2233224	578.986	19
	HOST__fv_crc_noinit__property	F	7051	504010	515.484	132	3071	1757519	236.065	17
	HOST__fv_timeout__property	O								
	SLAVE__fv__property	O								
profisafe_o4_host	HOST__fv_crc__property	F	3691	1	566.197	7	1725	29632	242.828	1
	HOST__fv_crc_noinit__property	T	3854	1	588.544	7	1034	38058	40.145	1
	HOST__fv_timeout__property	T	3634	1	583.174	7	545	1	7.538	0
profisafe_o4_slave	SLAVE__fv__property	T	501	1	5.961	3	531	1	9.932	1
profisafe_o5	fv__host_crc__3__property	O								
	fv__host_crc__3r__property	O								
	fv__host_to__3__property	O								
	fv__host_to__3r__property	O								
	fv__slave_crc__3__property	O								
	fv__slave_crc__3r__property	O								
	fv__slave_to__3__property	O								
	fv__slave_to__3r__property	O								
profisafe_o5_host	HOST__fv_crc__property	F	4221	1	1180.446	8	1201	1	94.974	2
	HOST__fv_crc_noinit__property	T	4854	1	1262.445	9	1028	1	35.130	2
	HOST__fv_timeout__property	T	4079	1	1094.131	8	596	1	9.009	0
profisafe_o6	fv__host_crc__3__property	O								
	fv__host_crc__3r__property	O								
	fv__host_to__3__property	O								
	fv__host_to__3r__property	O								
	fv__slave_crc__3__property	O								
	fv__slave_crc__3r__property	O								
	fv__slave_to__3__property	O								
	fv__slave_to__3r__property	O								
profisafe_o6_host	HOST__fv_crc__property	F	4616	1	2062.622	11	1418	1	217.808	2
	HOST__fv_crc_noinit__property	T	5828	1	2332.349	14	1195	1	65.899	2
	HOST__fv_timeout__property	T	4456	1	2031.426	11	733	1	14.293	0

Table A13: BDD Language Inclusion Check with Heuristic maxL and minS
for PROFIsafe Output-Slave Models
Projection limit 1000

Model	Property		MinT				MinCut			
			States	Nodes	Time(s)	O	States	Nodes	Time(s)	O
profisafe_o4	HOST__fv_crc__property	F	4448	1409334	391.026	22	2955	1695209	235.284	30
	HOST__fv_crc_noinit__property	F	4179	892506	349.852	22	3211	1704409	235.663	30
	HOST__fv_timeout__property	O								
	SLAVE__fv__property	O								
profisafe_o4_host	HOST__fv_crc__property	F	1021	1	50.479	2	552	1	10.804	0
	HOST__fv_crc_noinit__property	T	1079	37580	79.855	1	953	1	31.522	0
	HOST__fv_timeout__property	T	543	1	12.050	0	443	1	6.468	0
profisafe_o4_slave	SLAVE__fv__property	T	1319	1	68.190	0	531	1	11.262	0
profisafe_o5	fv__host_crc__3__property	O								
	fv__host_crc__3r__property	O								
	fv__host_to__3__property	O								
	fv__host_to__3r__property	O								
	fv__slave_crc__3__property	O								
	fv__slave_crc__3r__property	O								
	fv__slave_to__3__property	O								
	fv__slave_to__3r__property	O								
profisafe_o5_host	HOST__fv_crc__property	F	1197	1	109.903	1	648	1	19.997	0
	HOST__fv_crc_noinit__property	T	1082	1	56.726	2	1079	1	51.118	0
	HOST__fv_timeout__property	T	595	1	13.681	0	497	1	9.162	0
profisafe_o6	fv__host_crc__3__property	O								
	fv__host_crc__3r__property	O								
	fv__host_to__3__property	O								
	fv__host_to__3r__property	O								
	fv__slave_crc__3__property	O								
	fv__slave_crc__3r__property	O								
	fv__slave_to__3__property	O								
	fv__slave_to__3r__property	O								
profisafe_o6_host	HOST__fv_crc__property	F	978	38452	79.037	3	782	1	33.678	0
	HOST__fv_crc_noinit__property	T	1054	1	70.573	2	1247	1	88.199	0
	HOST__fv_timeout__property	T	734	1	22.000	0	585	1	13.435	0

Table A14: BDD Language Inclusion Check with Heuristic minT and minCut
for PROFIsafe Output-Slave Models
Projection limit 1000

Model	Property		MaxL			MinS		
			States	Time(s)	O	States	Time(s)	O
profisafe_i4_host	HOST__fv_crc__property	FALSE	1	556.417	6	1	273.527	0
	HOST__fv_crc_noinit__property	TRUE	1	599.088	6	1	511.760	0
	HOST__fv_timeout__property	TRUE	1	606.392	6	1	6.866	0
profisafe_i4_slave	SLAVE__fv__property	TRUE	1	1135.698	3	1	1.383	0
profisafe_i4	HOST__fv_crc__property	FALSE	3407088	621.469	104	80187	254.657	11
	HOST__fv_crc_noinit__property	FALSE		O		80204	293.997	11
	HOST__fv_timeout__property	TRUE	3071	1085.144	92	1	153.574	9
	SLAVE__fv__property	TRUE	1544	529.883	147	1	123.263	9
profisafe_i5_host	HOST__fv_crc__property	FALSE	1	1204.148	6	132	515.312	1
	HOST__fv_crc_noinit__property	TRUE	1	1304.929	6	270	658.965	1
	HOST__fv_timeout__property	TRUE	1	1225.607	6	1	8.455	0
profisafe_i5	fv__host_crc__3__property	FALSE		O			O	
	fv__host_crc__3r__property	FALSE		O			O	
	fv__host_to__3__property	FALSE		O			O	
	fv__host_to__3r__property	FALSE		O			O	
	fv__slave_crc__3__property	FALSE	8552464	1946.125	86		O	
	fv__slave_crc__3r__property	FALSE	2688049	1301.392	91		O	
	fv__slave_to__3__property	FALSE	2892	1208.016	104	696	1424.764	25
	fv__slave_to__3r__property	FALSE	2877	1185.254	108		O	
profisafe_i6_host	HOST__fv_crc__property	FALSE	1	2502.716	6	148	1260.681	1
	HOST__fv_crc_noinit__property	TRUE	1	2318.840	7	1	307.208	2
	HOST__fv_timeout__property	TRUE	1	2389.893	6	1	13.118	0
profisafe_i6	fv__host_crc__3__property	FALSE		O			O	
	fv__host_crc__3r__property	FALSE		O			O	
	fv__host_to__3__property	FALSE		O			O	
	fv__host_to__3r__property	FALSE		O			O	
	fv__slave_crc__3__property	FALSE	8552464	3061.531	97		O	
	fv__slave_crc__3r__property	FALSE	2688049	2232.852	102		O	
	fv__slave_to__3__property	FALSE	2893	2135.731	118	883	3447.343	35
	fv__slave_to__3r__property	FALSE	2885	2322.494	121	3658	3447.343	42

Table A15: Modular Language Inclusion Check with Heuristic maxL and minS for PROFIsafe Input-Slave Models
Projection limit 3000

Model	Property		MinT			MinCut		
			States	Time(s)	O	States	Time(s)	O
profisafe_i4_host	HOST__fv_crc__property	FALSE	1	337.883	0	1	10.965	0
	HOST__fv_crc_noinit__property	TRUE	1	330.176	0	1	31.572	0
	HOST__fv_timeout__property	TRUE	1	9.261	0	1	6.602	0
profisafe_i4_slave	SLAVE__fv__property	TRUE	1	1.243	0	1	3.845	0
profisafe_i4	HOST__fv_crc__property	FALSE	98891	293.275	14	95901	652.839	28
	HOST__fv_crc_noinit__property	FALSE	1695848	377.892	13	95901	654.238	28
	HOST__fv_timeout__property	TRUE	1	243.121	18	1	62.272	34
	SLAVE__fv__property	TRUE	1	162.950	20		O(C)	
profisafe_i5_host	HOST__fv_crc__property	FALSE	1	940.612	0	1	18.233	0
	HOST__fv_crc_noinit__property	TRUE	704	77.988	1	1	53.077	0
	HOST__fv_timeout__property	TRUE	1	14.125	0	1	8.981	0
profisafe_i5	fv__host_crc__3__property	FALSE	15672713	342.320	23		O	
	fv__host_crc__3r__property	FALSE		O			O	
	fv__host_to__3__property	FALSE		O		4486920	118.322	39
	fv__host_to__3r__property	FALSE		O			O	
	fv__slave_crc__3__property	FALSE		O			O	
	fv__slave_crc__3r__property	FALSE		O			O	
	fv__slave_to__3__property	FALSE	3028	205.741	33	3016	461.820	50
	fv__slave_to__3r__property	FALSE		O		1123	189.456	48
profisafe_i6_host	HOST__fv_crc__property	FALSE	168	1546.556	1	1	33.475	0
	HOST__fv_crc_noinit__property	TRUE	792	323.527	1	1	80.437	0
	HOST__fv_timeout__property	TRUE	1	20.360	0	1	12.523	0
profisafe_i6	fv__host_crc__3__property	FALSE		O			O	
	fv__host_crc__3r__property	FALSE		O			O	
	fv__host_to__3__property	FALSE		O			O	
	fv__host_to__3r__property	FALSE	2173139	247.274	33		O	
	fv__slave_crc__3__property	FALSE		O			O	
	fv__slave_crc__3r__property	FALSE		O			O	
	fv__slave_to__3__property	FALSE	3028	359.776	44	3016	615.794	61
	fv__slave_to__3r__property	FALSE	3745	548.695	38	1230	320.412	60

Table A16: Modular Language Inclusion Check with Heuristic minT and minCut for PROFIsafe Input-Slave Models
Projection limit 3000

Model	Property		MaxL			MinS		
			States	Time(s)	O	States	Time(s)	O
profisafe_o4	HOST__fv_crc__property	FALSE		O			O	
	HOST__fv_crc_noinit__property	FALSE		O			O	
	HOST__fv_timeout__property	TRUE	1	560.932	148	1	121.834	15
	SLAVE__fv__property	TRUE	1	574.082	153	1	74.286	16
profisafe_o4_host	HOST__fv_crc__property	FALSE	1	568.506	6	1	303.905	0
	HOST__fv_crc_noinit__property	TRUE	1	663.112	6	1	515.460	0
	HOST__fv_timeout__property	TRUE	1	614.401	6	1	7.046	0
profisafe_o4_slave	SLAVE__fv__property	TRUE	1	20.200	3	1	52.107	0
profisafe_o5	fv__host_crc__3__property	OVERFLOW						
	fv__host_crc__3r__property	OVERFLOW						
	fv__host_to__3__property	OVERFLOW						
	fv__host_to__3r__property	OVERFLOW						
	fv__slave_crc__3__property	OVERFLOW						
	fv__slave_crc__3r__property	OVERFLOW						
	fv__slave_to__3__property	OVERFLOW						
	fv__slave_to__3r__property	OVERFLOW						
profisafe_o5_host	HOST__fv_crc__property	FALSE	1	1159.325	6	132	521.361	1
	HOST__fv_crc_noinit__property	TRUE	1	1194.144	6	270	748.167	1
	HOST__fv_timeout__property	TRUE	1	1269.770	6	1	8.377	0
profisafe_o6	fv__host_crc__3__property	OVERFLOW						
	fv__host_crc__3r__property	OVERFLOW						
	fv__host_to__3__property	OVERFLOW						
	fv__host_to__3r__property	OVERFLOW						
	fv__slave_crc__3__property	OVERFLOW						
	fv__slave_crc__3r__property	OVERFLOW						
	fv__slave_to__3__property	OVERFLOW						
	fv__slave_to__3r__property	OVERFLOW						
profisafe_o6_host	HOST__fv_crc__property	FALSE	1	2385.076	6	148	1365.328	1
	HOST__fv_crc_noinit__property	TRUE	1	2498.394	6	1	314.453	2
	HOST__fv_timeout__property	TRUE	1	2518.749	6	1	13.481	0

Table A17: Modular Language Inclusion Check with Heuristic maxL and minS for PROFIsafe Output-Slave Models
Projection limit 3000

Model	Property		MinT			MinCut		
			States	Time(s)	O	States	Time(s)	O
profisafe_o4	HOST__fv_crc__property	FALSE		O			O	
	HOST__fv_crc_noinit__property	FALSE		O			O	
	HOST__fv_timeout__property	TRUE	1	183.474	21	1	139.552	44
	SLAVE__fv__property	TRUE	1	310.151	22	1	83.078	46
profisafe_o4_host	HOST__fv_crc__property	FALSE	1	321.365	0	1	10.643	0
	HOST__fv_crc_noinit__property	TRUE	1	317.131	0	1	31.239	0
	HOST__fv_timeout__property	TRUE	1	9.724	0	1	6.810	0
profisafe_o4_slave	SLAVE__fv__property	TRUE	1	2.198	0	1	10.206	0
profisafe_o5	fv__host_crc__3__property	OVERFLOW						
	fv__host_crc__3r__property	OVERFLOW						
	fv__host_to__3__property	OVERFLOW						
	fv__host_to__3r__property	OVERFLOW						
	fv__slave_crc__3__property	OVERFLOW						
	fv__slave_crc__3r__property	OVERFLOW						
	fv__slave_to__3__property	OVERFLOW						
	fv__slave_to__3r__property	OVERFLOW						
profisafe_o5_host	HOST__fv_crc__property	FALSE	132	471.531	1	1	20.101	0
	HOST__fv_crc_noinit__property	TRUE	704	106.273	1	1	50.895	0
	HOST__fv_timeout__property	TRUE	1	13.715	0	1	9.378	0
profisafe_o6	fv__host_crc__3__property	OVERFLOW						
	fv__host_crc__3r__property	OVERFLOW						
	fv__host_to__3__property	OVERFLOW						
	fv__host_to__3r__property	OVERFLOW						
	fv__slave_crc__3__property	OVERFLOW						
	fv__slave_crc__3r__property	OVERFLOW						
	fv__slave_to__3__property	OVERFLOW						
	fv__slave_to__3r__property	OVERFLOW						
profisafe_o6_host	HOST__fv_crc__property	FALSE	148	1175.865	1	1	31.380	0
	HOST__fv_crc_noinit__property	TRUE	1	328.276	2	1	81.563	0
	HOST__fv_timeout__property	TRUE	1	23.098	1	1	12.992	0

Table A18: Modular Language Inclusion Check with Heuristic minT and minCut for PROFIsafe Output-Slave Models
Projection limit 3000

Model	Property		MaxL			MinS		
			States	Time(s)	O	States	Time(s)	O
profisafe_i4_host	HOST__fv_crc__property	FALSE	1	553.009	6	130	231.400	0
	HOST__fv_crc_noinit__property	TRUE	1	578.415	7	616	39.918	0
	HOST__fv_timeout__property	TRUE	1	624.839	6	1	7.084	0
profisafe_i4_slave	SLAVE__fv__property	TRUE	1	227.903	5	1	1.380	0
profisafe_i4	HOST__fv_crc__property	FALSE	5828428	619.767	94	1765774	58.474	14
	HOST__fv_crc_noinit__property	FALSE		O		164497	187.469	11
	HOST__fv_timeout__property	TRUE	1709	447.669	87	1	15.850	9
	SLAVE__fv__property	TRUE	7404	400.329	73	1	15.725	10
profisafe_i5_host	HOST__fv_crc__property	FALSE	1	1126.293	7	1	96.860	2
	HOST__fv_crc_noinit__property	TRUE	1	1205.904	8	1	38.787	2
	HOST__fv_timeout__property	TRUE	1	1041.700	7	1	8.096	0
profisafe_i5	fv__host_crc__3__property	FALSE		O			O	
	fv__host_crc__3r__property	FALSE		O			O	
	fv__host_to__3__property	FALSE		O		5782362	146.083	26
	fv__host_to__3r__property	FALSE		O			O	
	fv__slave_crc__3__property	FALSE	34003378	1174.377	84		O	
	fv__slave_crc__3r__property	FALSE		O			O	
	fv__slave_to__3__property	FALSE	660	722.084	103	598	53.437	24
	fv__slave_to__3r__property	FALSE	786	738.386	103		O	
profisafe_i6_host	HOST__fv_crc__property	FALSE		O		1	196.907	2
	HOST__fv_crc_noinit__property	TRUE		O		1	60.065	2
	HOST__fv_timeout__property	TRUE		O		1	13.485	0
profisafe_i6	fv__host_crc__3__property	FALSE		O			O	
	fv__host_crc__3r__property	FALSE		O			O	
	fv__host_to__3__property	FALSE		O		8086730	235.899	37
	fv__host_to__3r__property	FALSE		O			O	
	fv__slave_crc__3__property	FALSE	14785501	2462.032	120		O	
	fv__slave_crc__3r__property	FALSE		O			O	
	fv__slave_to__3__property	FALSE	8289	2151.806	121	6872	85.735	36
	fv__slave_to__3r__property	FALSE	13781	2126.184	131		O	

Table A19: Modular Language Inclusion Check with Heuristic maxL and minS for PROFIsafe Input-Slave Models
Projection limit 1000

Model	Property		MinT			MinCut		
			States	Time(s)	O	States	Time(s)	O
profisafe_i4_host	HOST__fv_crc__property	FALSE	130	260.716	1	1	11.251	0
	HOST__fv_crc_noinit__property	TRUE	616	65.558	2	1	30.171	0
	HOST__fv_timeout__property	TRUE	1	11.723	1	1	6.672	0
profisafe_i4_slave	SLAVE__fv__property	TRUE	1	1.490	1	1	3.804	0
profisafe_i4	HOST__fv_crc__property	FALSE	216958	259.715	13	60834	248.216	27
	HOST__fv_crc_noinit__property	FALSE	1067974	277.337	12	60834	256.500	27
	HOST__fv_timeout__property	TRUE	1	83.446	17	1	42.758	34
	SLAVE__fv__property	TRUE		O			O(C)	
profisafe_i5_host	HOST__fv_crc__property	FALSE	204	61.280	15	1	19.836	0
	HOST__fv_crc_noinit__property	TRUE	1	41.184	2	1	56.323	0
	HOST__fv_timeout__property	TRUE	1	13.744	1	1	8.857	0
profisafe_i5	fv__host_crc__3__property	FALSE		O			O	
	fv__host_crc__3r__property	FALSE		O			O	
	fv__host_to__3__property	FALSE	8586721	233.289	29	9023488	142.780	53
	fv__host_to__3r__property	FALSE		O		8852459	154.914	39
	fv__slave_crc__3__property	FALSE		O		1562	51.883	46
	fv__slave_crc__3r__property	FALSE		O			O	
	fv__slave_to__3__property	FALSE	3028	156.759	34	2949	49.731	45
	fv__slave_to__3r__property	FALSE		O		4367	261.913	47
profisafe_i6_host	HOST__fv_crc__property	FALSE	1	207.260	1	1	32.801	0
	HOST__fv_crc_noinit__property	TRUE	1	80.116	1	1	91.934	0
	HOST__fv_timeout__property	TRUE	1	19.229	0	1	12.771	0
profisafe_i6	fv__host_crc__3__property	FALSE		O			O	
	fv__host_crc__3r__property	FALSE		O			O	
	fv__host_to__3__property	FALSE		O		17315058	290.487	66
	fv__host_to__3r__property	FALSE		O			O	
	fv__slave_crc__3__property	FALSE		O			O	
	fv__slave_crc__3r__property	FALSE		O			O	
	fv__slave_to__3__property	FALSE	3020	182.092	35	13898	39.329	55
	fv__slave_to__3r__property	FALSE	3767	225.012	42	20221	312.680	57

Table A20: Modular Language Inclusion Check with Heuristic minT and minCut for PROFIsafe Input-Slave Models
Projection limit1000

Model	Property		MaxL			MinS		
			States	Time(s)	O	States	Time(s)	O
profisafe_o4	HOST__fv_crc__property	FALSE		O			O	
	HOST__fv_crc_noinit__property	FALSE		O			O	
	HOST__fv_timeout__property	TRUE	1	467.74	137	1	46.059	16
	SLAVE__fv__property	TRUE	3821	410.454	116	1	52.713	16
profisafe_o4_host	HOST__fv_crc__property	FALSE	1	595.702	7	130	248.098	1
	HOST__fv_crc_noinit__property	TRUE	1	584.297	7	616	39.459	1
	HOST__fv_timeout__property	TRUE	1	529.500	7	1	7.280	0
profisafe_o4_slave	SLAVE__fv__property	TRUE	1	5.943	3	1	10.420	0
profisafe_o5	fv__host_crc__3__property	OVERFLOW						
	fv__host_crc__3r__property	OVERFLOW						
	fv__host_to__3__property	OVERFLOW						
	fv__host_to__3r__property	OVERFLOW						
	fv__slave_crc__3__property	OVERFLOW						
	fv__slave_crc__3r__property	OVERFLOW						
	fv__slave_to__3__property	OVERFLOW						
	fv__slave_to__3r__property	OVERFLOW						
profisafe_o5_host	HOST__fv_crc__property	FALSE	1	1102.754	8	1	92.165	2
	HOST__fv_crc_noinit__property	TRUE	1	1276.276	9	1	38.803	2
	HOST__fv_timeout__property	TRUE	1	1119.956	8	1	8.362	0
profisafe_o6	fv__host_crc__3__property	OVERFLOW						
	fv__host_crc__3r__property	OVERFLOW						
	fv__host_to__3__property	OVERFLOW						
	fv__host_to__3r__property	OVERFLOW						
	fv__slave_crc__3__property	OVERFLOW						
	fv__slave_crc__3r__property	OVERFLOW						
	fv__slave_to__3__property	OVERFLOW						
	fv__slave_to__3r__property	OVERFLOW						
profisafe_o6_host	HOST__fv_crc__property	FALSE	1	1993.971	11	1	194.342	2
	HOST__fv_crc_noinit__property	TRUE	1	2393.661	14	1	64.977	2
	HOST__fv_timeout__property	TRUE	1	1979.234	11	1	13.645	0

Table A21: Modular Language Inclusion Check with Heuristic maxL and minS for PROFIsafe Output-Slave Models
Projection limit 1000

Model	Property		MinT			MinCut		
			States	Time(s)	O	States	Time(s)	O
profisafe_o4	HOST__fv_crc__property	FALSE		O			O	
	HOST__fv_crc_noinit__property	FALSE		O			O	
	HOST__fv_timeout__property	TRUE	1	162.954	25	1	63.378	43
	SLAVE__fv__property	TRUE	4790	182.378	24	1	54.574	45
profisafe_o4_host	HOST__fv_crc__property	FALSE	130	239.675	1	1	10.935	0
	HOST__fv_crc_noinit__property	TRUE	616	84.321	2	1	32.347	0
	HOST__fv_timeout__property	TRUE	1	12.786	0	1	6.726	0
profisafe_o4_slave	SLAVE__fv__property	TRUE	1	63.830	0	1	11.216	0
profisafe_o5	fv__host_crc__3__property	OVERFLOW						
	fv__host_crc__3r__property	OVERFLOW						
	fv__host_to__3__property	OVERFLOW						
	fv__host_to__3r__property	OVERFLOW						
	fv__slave_crc__3__property	OVERFLOW						
	fv__slave_crc__3r__property	OVERFLOW						
	fv__slave_to__3__property	OVERFLOW						
	fv__slave_to__3r__property	OVERFLOW						
profisafe_o5_host	HOST__fv_crc__property	FALSE	204	69.530	1	1	19.774	0
	HOST__fv_crc_noinit__property	TRUE	1	59.421	3	1	50.439	0
	HOST__fv_timeout__property	TRUE	1	14.548	1	1	8.829	0
profisafe_o6	fv__host_crc__3__property	OVERFLOW						
	fv__host_crc__3r__property	OVERFLOW						
	fv__host_to__3__property	OVERFLOW						
	fv__host_to__3r__property	OVERFLOW						
	fv__slave_crc__3__property	OVERFLOW						
	fv__slave_crc__3r__property	OVERFLOW						
	fv__slave_to__3__property	OVERFLOW						
	fv__slave_to__3r__property	OVERFLOW						
profisafe_o6_host	HOST__fv_crc__property	FALSE	1	210.643	2	1	32.950	0
	HOST__fv_crc_noinit__property	TRUE	1	78.894	1	1	88.229	0
	HOST__fv_timeout__property	TRUE	1	22.831	0	1	13.131	0

Table A22: Modular Language Inclusion Check with Heuristic minT and minCut for PROFIsafe Output-Slave Models
Projection limit 1000